Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps

Kevin Moran, *Student Member, IEEE,* Carlos Bernal-Cárdenas, *Student Member, IEEE,*
Michael Curcio, *Student Member, IEEE,* Richard Bonett, *Student Member, IEEE,*
and Denys Poshyvanyk, *Member, IEEE*

Abstract—It is common practice for developers of user-facing software to transform a mock-up of a graphical user interface (GUI) into code. This process takes place both at an application's inception and in an evolutionary context as GUI changes keep pace with evolving features. Unfortunately, this practice is challenging and time-consuming. In this paper, we present an approach that automates this process by enabling accurate prototyping of GUIs via three tasks: *detection*, *classification*, and *assembly*. First, logical components of a GUI are *detected* from a mock-up artifact using either computer vision techniques or mock-up metadata. Then, software repository mining, automated dynamic analysis, and deep convolutional neural networks are utilized to accurately *classify* GUI-components into domain-specific types (*e.g.*, toggle-button). Finally, a data-driven, K-nearest-neighbors algorithm generates a suitable hierarchical GUI structure from which a prototype application can be automatically *assembled*. We implemented this approach for Android in a system called REDRAW. Our evaluation illustrates that REDRAW achieves an average GUI-component classification accuracy of 91% and assembles prototype applications that closely mirror target mock-ups in terms of visual affinity while exhibiting reasonable code structure. Interviews with industrial practitioners illustrate ReDraw's potential to improve real development workflows.

Index Terms—GUI, CNN, Mobile, Prototyping, Machine-Learning, Mining Software Repositories.

✦

1 INTRODUCTION

MOST modern user-facing software applications are GUI-centric, and rely on attractive user interfaces (UI) and intuitive user experiences (UX) to attract customers, facilitate the effective completion of computing tasks, and engage users. Software with cumbersome or aesthetically displeasing UIs are far less likely to succeed, particularly as companies look to differentiate their applications from competitors with similar functionality. This phenomena can be readily observed in mobile application marketplaces such as the App Store [1], or Google Play [2], where many competing applications (also known as *apps*) offering similar functionality (*e.g.*, task managers, weather apps) largely distinguish themselves via UI/UX [3]. Thus, an important step in developing any GUI-based application is drafting and prototyping design mock-ups, which facilitates the instantiation and experimentation of UIs in order to evaluate or prove-out abstract design concepts. In industrial settings with larger teams, this process is typically carried out by dedicated designers who hold domain specific expertise in crafting attractive, intuitive GUIs using image-editing software such as Photoshop [4] or Sketch [5]. These design teams are often responsible for expressing a coherent design language across the many facets of a company's digital presence, including websites, applications and digital marketing materials. Some components of this design process also tend to carry over to smaller independent development teams who practice design or prototyping processes by creating wireframes or mock-ups to judge design ideas before com-

mitting to spending development resources implementing them. After these initial design drafts are created it is critical that they are faithfully translated into code in order for the end-user to experience the design and user interface in its intended form.

This process (which often involves multiple iterations) has been shown by past work and empirical studies to be challenging, time-consuming, and error prone [6], [7], [8], [9], [10] particularly if the design and implementation are carried out by different teams (which is often the case in industrial settings [10]). Additionally, UI/UX teams often practice an iterative design process, where feedback is collected regarding the effectiveness of GUIs at early stages. Using prototypes would be preferred, as more detailed feedback could be collected; however, with current practices and tools this is typically too costly [11], [12]. Furthermore, past work on detecting GUI design violations in mobile apps highlights the importance of this problem from an industrial viewpoint [10]. According to a study conducted with Huawei, a major telecommunications company, 71 unique application screens containing 82 design violations resulting from the company's iterative design and development process were empirically categorized using a grounded-theory approach. This resulted in a taxonomy of mobile design violations spanning three major categories and 14 subcategories and illustrates the difficulties developers can have faithfully implementing GUIs for mobile apps as well as the burden that design violations introduced by developers can place on the overarching development process.

Many fast-moving startups and fledgling companies attempting to create software prototypes in order to demonstrate ideas and secure investor support would also greatly benefit from rapid application prototyping. Rather than

---

- *All authors are with the Department of Computer Science, College of William & Mary, Williamsburg, VA, 23185.*
  E-mail: {kpmoran, cebernal, mjcurcio, rfbonett, denys}@cs.wm.edu

spending scarce time and resources on iteratively designing and coding user interfaces, an accurate automated approach would likely be preferred. This would allow smaller companies to put more focus on features and value and less on translating designs into workable application code. Given the frustrations that front-end developers and designers face with constructing accurate GUIs, there is a clear need for automated support.

To help mitigate the difficulty of this process, some modern IDEs, such as XCode [13], Visual Studio [14], and Android Studio [15], offer built-in GUI editors. However, recent research suggests that using these editors to create complex, high-fidelity GUIs is cumbersome and difficult [11], as users are prone to introducing bugs and presentation failures even for simple tasks [16]. Other commercial solutions include offerings for collaborative GUI-design and for interactive previewing of designs on target devices or browsers (displayed using a custom framework, with limited functionality) [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], but none offer an end-to-end solution capable of automatically translating a mock-up into accurate native code for a target platform. It is clear that an automated tool capable of even partially automating this process could significantly reduce the burden on the design and development processes.

Unfortunately, automating the prototyping process for GUIs is a difficult task. At the core of this difficulty is the need to bridge a broad abstraction gap that necessitates reasoning accurate user interface code from either pixel-based, graphical representations of GUIs or digital design sketches. Typically, this abstraction gap is bridged by a developer's domain knowledge. For example, a developer is capable of recognizing discrete objects in a mock-up that should be instantiated as components on the screen, categorizing them into proper categories based on their intended functionalities, and arranging them in a suitable hierarchical structure such that they display properly on a range of screen sizes. However, even for a skilled developer, this process can be time-consuming and prone to errors [10]. Thus, it follows that an approach which automates the GUI prototyping process must bridge the image-to-code abstraction gap. This, in turn, requires the creation of a model capable of representing the domain knowledge typically held by a developer, and applying this knowledge to create accurate prototypes.

Given that, within a single software domain, the design and functionality of GUIs can vary dramatically, it is unlikely that manually encoded information or heuristics would be capable of fully supporting such complex tasks. Furthermore, creating, updating, and maintaining such heuristics manually is a daunting task. Thus, we propose to learn this domain knowledge using a data-driven approach that leverages machine learning (ML) techniques and the GUI information already present in existing apps (specifically screenshots and GUI metadata) acquired via mining software repositories (MSR).

More specifically, we present an approach that deconstructs the prototyping process into the tasks of: *detection*, *classification*, and *assembly*. The first task involves *detecting* the bounding boxes of atomic elements (*e.g.*, GUI-components which cannot be further decomposed) of a user interface from a mock-up design artifact, such as pixel-based images. This challenge can be solved either by parsing information regarding objects representing GUI-components directly from mock-up artifacts (*e.g.*, parsing exported metadata from Photoshop), or using CV techniques to infer objects [8]. Once the GUI-components from a design artifact have been identified, they need to be *classified* into their proper domain-specific types (*e.g.*, button, dropdown menu, progress bar). This is, in essence, an image classification task, and research on this topic has shown tremendous progress in recent years, mainly due to advancements in deep convolutional neural networks (CNNs) [29], [30], [31], [32], [33]. However, because CNNs are a supervised learning technique, they typically require a large amount of training data, such as the ILSVRC dataset [34], to be effective. We assert that automated dynamic analysis of applications mined from software repositories can be applied to collect screenshots and GUI metadata that can be used to *automatically* derive labeled training data. Using this data, a CNN can be effectively trained to classify images of GUI-Components from a mock-up (extracted using the detected bounding boxes) into their domain specific GUI-component types. However, classified images of components are not enough to *assemble* effective GUI code. GUIs are typically represented in code as hierarchal trees, where logical groups of components are bundled together in containers. We illustrate that an iterative K-nearest-neighbors (KNN) algorithm and CV techniques operating on mined GUI metadata and screenshots can construct realistic hierarchies of GUI-components that can be translated into code.

We have implemented the approach described above in a system called REDRAW for the Android platform. We mined 8,667 of the top-rated apps from the Google Play, executed these apps using a fully automated input generation approach (*e.g.*, GUI-ripping) derived from our prior work on mobile testing [35], [36], and extracted the GUI-hierarchies for the most popular screens from each app. We then trained a CNN on the most popular native Android GUI-component types as observed in the mined screens. REDRAW uses this classifier in combination with an iterative KNN algorithm and additional CV techniques to translate different types of mock-up artifacts into prototype Android apps. We performed a comprehensive set of three studies evaluating REDRAW aimed at measuring (i) the accuracy of the CNN-based classifier (measured against a baseline feature descriptor and Support Vector Machine based technique), (ii) the similarity of generated apps to the mock-up artifacts (both visually and structurally), and (iii) the potential industrial applicability of our system, through semi-structured interviews with mobile designers and developers at Google, Huawei and Facebook. Our results show that our CNN-based GUI-component classifier achieves a top-1 average precision of 91% (*i.e.*, when the top class predicted by the CNN is correct), our generated applications share high visual similarity to their mock-up artifacts, the code structure for generated apps is similar to that of real applications, and REDRAW has the potential to improve and facilitate the prototyping and development of mobile apps with some practical extensions. Our evaluation also illustrates how REDRAW outperforms other related approaches for mobile application prototyping, REMAUI [8]

and pix2code [37]. Finally, we provide a detailed discussion of the limitations of our approach and promising avenues for future research that build upon the core ideas presented.

In summary, our paper makes the following noteworthy contributions:

- The introduction of a novel approach for prototyping software GUIs rooted in a combination of techniques drawn from program analysis, MSR, ML, and CV; and an implementation of this approach in a tool called REDRAW for the Android platform;
- A comprehensive empirical evaluation of REDRAW, measuring several complimentary quality metrics, offering comparison to related work, and describing feedback from industry professionals regarding its utility;
- An online appendix [38] showcasing screenshots of generated apps and study replication information;
- As part of implementing REDRAW we collected the largest known dataset of mobile application GUI data containing screenshots and GUI related metadata for over 14k screens and over 190k GUI-components.
- Publicly available open source versions of the REDRAW code, datasets, and trained ML models [38][1].

2 BACKGROUND & RELATED WORK

In this section we introduce concepts related to the mock-up driven development process referenced throughout the paper and survey related work, distilling the novelty of our approach in context.

2.1 Background & Problem Statement

The first concept of a mock-up driven development practice we reference in this paper is that of *mock-up artifacts*, which we define as:

Definition 1 - **Mock-Up Artifact:** *An artifact of the software design and development process which stipulates design guidelines for GUIs and its content.*

In industrial mobile app development, mock-up artifacts typically come in the form of high fidelity images (with or without meta-data) created by designers using software such as Photoshop [4] or Sketch [5]. In this scenario, depending on design and development workflows, metadata containing information about the constituent parts of the mock-up images can be exported and parsed from these artifacts [2]. Independent developers may also use screenshots of existing apps to prototype their own apps. In this scenario, in addition to screenshots of running applications, runtime GUI-information (such as the html DOM-tree of a web app or the GUI-hierarchy of a mobile app) can be extracted to further aid in the prototyping process. However, this is typically *not* possible in the context of mock-up driven development (which our approach aims to support), as executable apps do not exist.

The second concept we define is that of *GUI-components* (also commonly called *GUI-widgets*). In this paper, we use the terms *GUI-component* and *component* interchangeably. We define these as:

Definition 2 - **GUI-Component:** *Atomic graphical elements with pre-defined functionality, displayed within a GUI of a software application.*

GUI-components have one of several domain dependent types, with each distinct type serving a different functional or aesthetic purpose. For example, in web apps common component types include dropdown menus and check-boxes, just to name a few.

The notion of *atomicity* is important in this definition, as it differentiates GUI-components from *containers*. The third concept we define is that of a *GUI-container*:

Definition 3 - **GUI-Container:** *A logical construct that groups member GUI-components and typically defines spatial display properties of its members.*

In modern GUI-centric apps, GUI-components are rarely rendered on the screen using pre-defined coordinates. Instead, logical groupings of containers form hierarchical structures (or *GUI-hierarchies*). These hierarchies typically define spatial information about their constituent components, and in many cases react to changes in the size of the display area (*i.e., reactive design*) [39]. For instance, a GUI-component that displays text may span the text according to the dimensions of its container.

Given these definitions, the problem that we aim to solve in this paper is the following:

Problem Statement: *Given a mock-up artifact, generate a prototype application that closely resembles the mock-up GUI both visually, and in terms of expected structure of the GUI-hierarchy.*

As we describe in Sec. 3, this problem can be broken down into three distinct tasks including the *detection* and *classification* of GUI-components, and the *assembly* of a realistic GUI-hierarchy and related code. In the scope of this paper, we focus on automatically generating GUIs for mobile apps that are visually and structurally similar (in terms of their GUI hierarchy). We leave the prototyping of interactive behaviors as future work and discuss this in Sec. 7.

2.2 Related Work

2.2.1 Reverse Engineering Mobile User Interfaces:

The most closely related research to the approach proposed in this paper is REMAUI, which aims to reverse engineer mobile app GUIs [8]. REMAUI uses a combination of Optical Character Recognition (OCR), CV, and mobile specific heuristics to detect components and generate a static app. The CV techniques utilized in REMAUI are powerful, and we build upon these innovations. However, REMAUI has key limitations compared to our work including: (i) it does not support the classification of detected components into their native component types and instead uses a binary classification of either text or images, limiting the real-world applicability of the approach, and (ii) it is unclear if the GUI-hierarchies generated by REMAUI are realistic or useful from a developer's point of view, as the GUI-hierarchies of the approach were not evaluated.

In comparison, the approach presented in this paper (i) is not specific to any particular domain (although we

---

1. Made available upon acceptance of the paper
2. For example, by exporting Scalable Vector Graphics (`.svg`) or `html` formats from Photoshop.

implement our approach for the Android platform as well) as we take a data-driven approach for classifying and generating GUI-hierarchies, (ii) is capable of classifying GUI-components into their respective types using CNNs, and (iii) is able to produce realistic GUI-hierarchies using a data-driven, iterative KNN algorithm in combination with CV techniques. In our evaluation, we offer a comparison of REDRAW to the REMAUI approach according to different quality attributes.

In addition to REMAUI, an open access paper (*i.e.*, non-peer-reviewed) was recently posted that implements an approach called pix2code [37], which shares common goals with the research we present in this paper. Namely, the authors implement an encoder/decoder model that they trained on information from ui-metadata and screenshots to translate target screenshots first into a domain specific language (DSL) and then into UI code. However, this approach exhibits several shortcomings that call into question the real-world applicability of the approach: (i) the approach was only validated on a small set of synthetically generated applications, and no large-scale user interface mining was performed; (ii) the approach requires a DSL which will need to be maintained and updated over time, adding to the complexity and effort required to utilize the approach in practice. Thus, it is difficult to judge how well the approach would perform on real UI data. In contrast, REDRAW is trained off a large scale dataset collected through a novel application of automated dynamic analysis for user interface mining. The data-collection and training process can be performed completely automatically and iteratively over time, helping to ease the burden of use for developers. To make for a complete comparison to current research-oriented approaches, we also include a comparison of the prototyping capability for real applications between REDRAW to the pix2code approach in Sections 4 & 5.

2.2.2 Other GUI-Design and Reverse Engineering Tools:
Given the prevalence of GUI-centric software, there has been a large body of work dedicated to building advanced tools to aid in the construction of GUIs and related code [40], [41], [42], [43], [44], [45], [46] and to reverse engineer GUIs [47], [48], [49], [50], [51], [52]. While these approaches are aimed at various goals, they all attempt to reason logical, or programatic info from graphical representations of GUIs.

However, these solutions exhibit one or more of the following shortcomings: (i) they rely on predefined models of GUIs to detect varying types of UI elements [48], [49], [50], [51], [52], (ii) they force designers or developers to compromise their workflow by imposing restrictions on how applications are designed or coded [40], [41], [42], [43], [44], [45] or (iii) they rely purely on reverse engineering existing apps using runtime information, which is not possible in the context of mock-up driven development. [45], [47]. These shortcomings can severely limit the applicability of such approaches. Given the large number of different types and styles of GUIs today, approaches requiring pre-defined models of GUI elements would be expensive to create and maintain, and will not scale or gain adoption across domains. Approaches that tie developers or designers into strict workflows (such as restricting the ways mock-ups are created or coded) struggle to gain adoption due to the competing flexibility of established image-editing software and coding platforms. Finally, approaches requiring runtime information of a *target* app cannot be used in a typical mock-up driven development scenario, as implementations do not exist yet. While our approach relies on runtime data, it is collected and processed *independently* of the target app or mock-up artifact. Our approach aims to overcome this list of shortcomings by leveraging MSR and ML techniques to automatically infer models of GUIs for different domains, and has the potential of integrating into current design and development workflows as illustrated in Sec. 5.4.

In addition to research on this topic, there are several commercial solutions which aim to improve the mock-up and prototyping process for different types of applications [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. These approaches allow for better collaboration among designers, and some more advanced offerings enable limited-functionality prototypes to be displayed on a target platform with support of a software framework. For instance, some tools will display screenshots of mock-ups on a mobile device through a preinstalled app, and allow designers to preview designs. However, these techniques *are not* capable of translating mock-up artifacts into GUI code, and tie designers into a specific, potentially less flexible software or service.

2.2.3 Image Classification using CNNs:
Large scale image recognition and classification has seen tremendous progress mainly due to advances in CNNs [29], [30], [31], [32], [33], [53]. These supervised ML approaches are capable of automatically learning robust, salient features of image categories from large numbers of labeled training images such as the ILSVRC dataset [34]. Building on top of LeCun's pioneering work [53], the first approach to see a significant performance improvement over existing techniques (that utilized predefined feature extraction) was AlexNet [29], which achieved a top-5 mean average error (MAE) of $\approx 15\%$ on ILSVRC12. The architecture for this network was relatively shallow, but later work would show the benefits and tradeoffs of using deeper architectures. Zeiler and Fergus developed the ZFNet [30] architecture which was able to achieve a lower MAE than AlexNet ($\approx 11\%$) and devised a methodology for visualizing the hidden layers (or activation maps) of CNNs. More recent approaches such as GoogLeNet [32] and Microsoft's ResNet [33] use deeper architectures (e.g., 22 and 152 layers respectively) and have managed to surpass human levels of accuracy on image classification tasks. However, the gains in network learning capacity afforded by deeper architectures come with a trade off in terms of training data requirements and training time. In this paper, we show that a relatively simple CNN architecture can be trained in a reasonable amount of time on popular classes of Android GUI-components, achieving a top-1 average classification accuracy of 91%.

3 APPROACH DESCRIPTION
We describe our approach for GUI prototyping around the three major phases of the process: *detection*, *classification*, & *assembly*. Fig. 1 illustrates an overview of the process that we will refer to throughout the description of the approach.
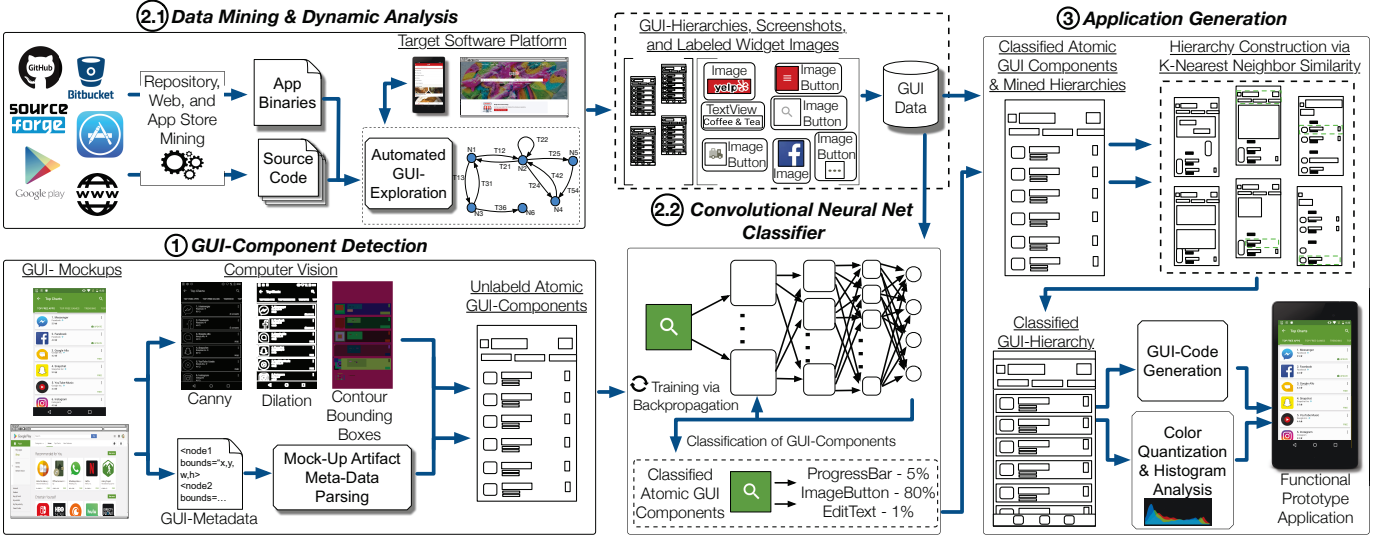
Fig. 1: Overview of Proposed Approach for Automated GUI-Prototyping

At a high-level, our approach first *detects* GUI-components from a mock-up artifact by either utilizing CV techniques or parsing meta-data directly from mock-up artifacts generated using professional photo-editing software. Second, to *classify* the detected GUI-components into proper types, we propose to train a CNN using GUI data gleaned from large-scale automated dynamic analysis of applications extracted by mining software repositories. The trained CNN can then be applied to mock-up artifacts to classify detected components. Finally, to construct a suitable GUI-hierarchy (*e.g.*, proper groupings of GUI-components in GUI-containers) we utilize a KNN-based algorithm that leverages the GUI-information extracted from the large-scale dynamic analysis to *assemble* a realistic nested hierarchy of GUI-components and GUI-containers. It is important to note that while we implement REDRAW for the Android platform, there are no fundamental engineering limitations that prevent our proposed methodology from being adapted to other platforms, such as the web or iOS. Thus, to illustrate our general approach, for each phase we first describe the proposed methodology and design decisions at high level and then discuss the implementation details specific to our instantiation of REDRAW for the Android platform.

3.1 Phase 1 - Detection of GUI-Components

The first task required of a GUI-prototyping approach is detecting the GUI-components that exist in a mock-up artifact. The main *goal* of this phase is to accurately infer the bounding boxes of atomic GUI-component elements (in terms of pixel-based coordinates) from a mock-up artifact. This allows for the cropping and extracting of individual images of GUI-components that will be utilized in the later stages of the prototyping process. This phase can be accomplished via one of two methodologies: (i) parsing data from mock-up artifacts, or (ii) using CV techniques to detect GUI-components. A visualization of this phase is illustrated in Fig. 1-①. In the following subsections we describe the *detection* procedure for both of these methodologies as well as our specific implementation within REDRAW.

3.1.1 Parsing Data from Design Mockups

The first method for detecting the GUI-components that exist in a mock-up artifact, shown in the bottom portion of Fig. 1-①, is to utilize the information encoded into mock-up artifacts. Given the importance of UI/UX in today's consumer facing software, many designers and small teams of developers work with professional grade image editing software, such as Photoshop [4] or Sketch [5] to create either wireframe or pixel perfect static images of GUIs that comprise mock-up artifacts. During this process photo-editing or design software is typically used to create a blank canvas with dimensions that match a target device screen or display area (with some design software facilitating scaling to multiple screen sizes [4], [5]). Then, images representing GUI-components are placed as editable objects on top of this canvas to construct the mock-up. Most of these tools are capable of exporting the mock-up artifacts in formats that encode spatial information about the objects on the canvas, such as using the Scalable Vector Graphics (.svg) format or html output [54]. Information about the layouts of objects, including the bounding boxes of these objects, can be parsed from these output formats, resulting in highly accurate detection of components. Therefore, if this metadata for the mock-up artifacts is available, it can be parsed to obtain extremely accurate bounding boxes for GUI-components that exist in a mock-up artifact which can then be utilized in the remainder of the prototyping process.

Given the spatial information encoded in metadata that is sometimes available in mock-up artifacts, one may question whether this information can also be used to reconstruct a hierarchical representation of GUI-components that could later aid in the code conversion process. Unfortunately, realistic *GUI-hierarchies* typically cannot be feasibly parsed from such artifacts for at least the following two reasons: (i) designers using photo-editing software to create mock-ups tend to encode a different hierarchal structure than a developer would, due to a designer lacking knowledge regarding the best manner in which to programmatically arrange GUI-components on a screen [10]; (ii) limitations

in photo-editing software can prohibit the creation of programmatically proper spatial layouts. Thus, any hierarchical structure parsed out of such artifacts is likely to be specific to designers' preferences, or restricted based on the capabilities of photo-editing software, limiting applicability in our prototyping scenario. For example, a designer might not provide enough GUI-containers to create an effective reactive mobile layout, or photo-editing software might not allow for relative positioning of GUI-components that scale across different screen sizes.

3.1.2 *Using CV Techniques for GUI-component Detection:*
While parsing information from mock-ups results in highly accurate bounding boxes for GUI-components this info may not always be available, either due to limitations in the photo-editing software being used or differing design practices, such as digitally or physically sketching mockups using pen displays, tablets, or paper. In these cases, a mock-up artifact may consist only of an image, and thus CV techniques are needed to identify relevant GUI-component info. To support these scenarios, our approach builds upon the CV techniques from [8] to detect GUI-component bounding boxes. This process uses a series of different CV techniques (Fig. 1-①) to infer bounding boxes around objects corresponding to GUI components in an image. First, Canny's edge detection algorithm [55] is used to detect the edges of objects in an image. Then these edges are dilated to merge edges close to one another. Finally, the contours of those edges are used to derive bounding boxes around atomic GUI-components. Other heuristics for merging text-based components using Optical Character Recognition (OCR) are used to merge the bounding boxes of logical blocks of text (*e.g.*, rather than detecting each word as its own component, sentences and paragraphs of text are merged).

3.1.3 *ReDraw Implementation - GUI Component Detection*
In implementing REDRAW, to support the scenario where metadata can be gleaned from mock-ups for Android applications we target artifacts created using the Marketch [54] plugin for Sketch [5], which exports mock-ups as a combination of `html` & `javascript`. Sketch is popular among mobile developers and offers extensive customization through a large library of plugins [56]. REDRAW parses the bounding boxes of GUI-components contained within the exported Marketch files. To support the scenario where meta-data related to mock-ups is not available, REDRAW uses CV techniques to automatically infer the bounding boxes of components from a static image. To accomplish this, we re-implemented the approach described in [8]. Thus, the input to the GUI-component detection phase of REDRAW is either a screenshot and corresponding `marketch` file (to which the marketch parsing procedure is applied), or a single screenshot (to which CV-based techniques are applied). The end result of the GUI-component detection process is a set of bounding box coordinates situated within the original input screenshot and a collection of images cropped from the original screenshot according to the derived bounding boxes that depict atomic GUI-components. This information is later fed into a CNN to be classified into Android specific component types in Phase 2.2. It should be noted that only *GUI-components* are detected during this process. On the other hand *GUI-containers* and the corresponding GUI-hierarchy are constructed in the *assembly* phase described in Sec. 3.3.

3.2 Phase 2 - GUI-component Classification
Once the bounding boxes of atomic GUI-component elements have been *detected* from a mock-up artifact, the next step in the prototyping process is to *classify* cropped images of specific GUI components into their domain specific types. To do this, we propose a data-driven and ML-based approach that utilizes CNNs. As illustrated in Fig. 1-②.1 and Fig. 1- ②.2, this phase has two major parts: (i) large scale software repository mining and automated dynamic analysis, and (ii) the training and application of a CNN to classify images of GUI-components. In the following subsections we first discuss the motivation and implementation of the repository mining and dynamic analysis processes before discussing the rationale for using a CNN and our specific architecture and implementation within REDRAW.

3.2.1 *Phase 2.1 - Large-Scale Software Repository Mining and Dynamic Analysis*
Given their supervised nature and deep architectures, CNNs aimed at the image classification task require a large amount of training data to achieve precise classification. Training data for traditional CNN image classification networks typically consists of a large set of images labeled with their corresponding classes, where labels correspond to the primary subject in the image. Traditionally, such datasets have to be manually procured, wherein humans painstakingly label each image in the dataset. However, we propose a methodology that *automates* the creation of labeled training data consisting of images of specific GUI-components cropped from full screenshots and labels corresponding to their domain specific type (*e.g.*, Buttons, or Spinners in Android) using fully-automated dynamic program analysis.

Our key insight for this automated dynamic analysis process is the following: *during automated exploration of software mined from large repositories platform specific frameworks can be utilized to extract meta-data describing the GUI, which can then be transformed into a large labeled training set suitable for a CNN.* As illustrated in Fig. 1-②.1, this process can be automated by mining software repositories to extract executables. Then a wealth of research in automated input generation for GUI-based testing of applications can be used to automatically execute mined apps by simulating user-input. For instance, if the target is a mobile app, input generation techniques relying on random-based [57], [58], [59], [60], [61], systematic [35], [62], [63], [64], [65], model-based [36], [62], [64], [66], [67], [68], [69], or evolutionary [70], [71] strategies could be adopted for this task. As the app is executed, screenshots and GUI-related metadata can be automatically extracted for each unique observed screen or layout of an app. Other similar automated GUI-ripping or crawling approaches can also be adapted for other platforms such as the web [72], [73], [74], [75], [76]. Screenshots can be captured using third party software or utilities included with a target operating system. GUI-related metadata can be collected from a variety of sources including accessibility services [77], `html` DOM information, or UI-frameworks

such as the `uiautomator` [78]. The GUI-metadata and screenshots can then be used to extract sub-images of GUI-components with their labeled types parsed from the related metadata describing each screen. The underlying quality of the resulting dataset relates to how well the labels describe the type of GUI-components displayed on a screen. Given that many of the software UI-frameworks that would be utilized to mine such data pull their information directly from utilities that render application GUI-components on the screen, this information is likely to be highly accurate. However, there are certain situations where the information gleaned from these frameworks contains minor inaccuracies or irrelevant cases. We discuss these cases and steps that can be taken to mitigate them in Sec. 3.2.4.

3.2.2 ReDraw Implementation - Software Repository Mining and Automated Dynamic Analysis

To procure a large set of Android apps to construct our training, validation, and test corpora for our CNN we mined free apps from Google Play at scale. To ensure the representativeness and quality of the apps mined, we extracted all categories from the Google Play store as of June 2017. Then we filtered out any category that primarily consisted of games, as games tend to use non-standard types of GUI-components that cannot be automatically extracted. This left us with a total of 39 categories. We then used a Google Play API library [79] to download the top 240 `APKs` from each category, excluding duplicates that existed in more than one category. This resulted in a total of 8,667 unique `APKs` after accounting for duplicates cross-listed across categories.

To extract information from the mined `APKs`, we implemented a large-scale dynamic analysis engine, called the *Execution Engine* that utilizes a systematic automated input generation approach based on our prior work on CRASHSCOPE and MONKEYLAB [35], [36], [65], [80] to explore the apps and extract screenshots and GUI-related information for visited screens. More specifically, our systematic GUI-exploration navigates a target apps's GUI in a Depth-First-Search (DFS) manner to exercise tappable, long-tappable, and type-able (*e.g.*, capable of accepting text input) components. During the systematic exploration we used Android's `uiautomator` [78] to extract GUI-related info as `xml` files that describe the hierarchy and various properties of components displayed on a given screen, and the Android `screencap` utility to collect screenshots. The `uiautomator xml` files contain various attributes and properties of each GUI-component displayed on an Android application screen, including the bounding boxes (*e.g.*, precise location and area within the screen) and component types (*e.g.*, EditText, Toggle Button). These attributes allow for individual sub-images for each GUI-component displayed on a given screen to be extracted from the corresponding screenshot and automatically labeled with their proper type.

The implementation of our DFS exploration strategy utilizes a state machine model where states are considered unique app screens, as indicated by their activity name and displayed window (*e.g.*, dialog box) extracted using the `adb shell dumpsys window` command. To allow for feasible execution times across the more than $8.6k$ apps in our dataset while still exploring several app screens, we
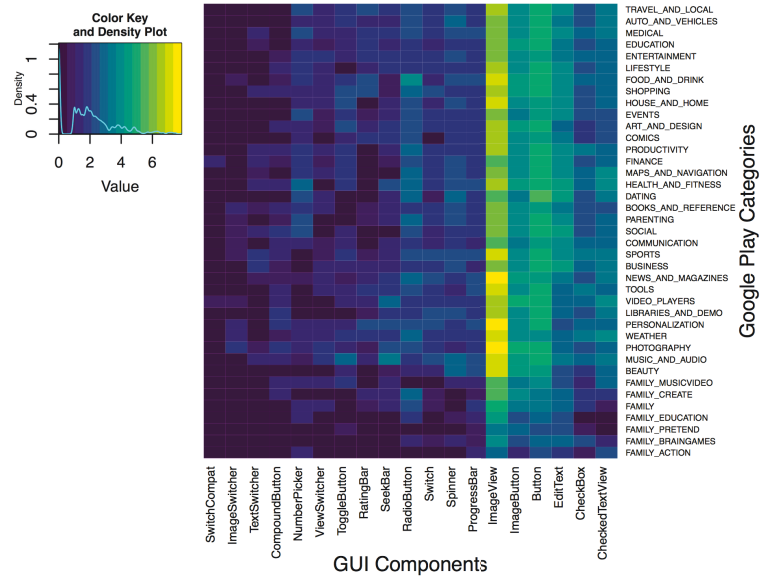

Fig. 2: Heat-map of GUI Components by Category

limited our exploration strategy to exercising 50 actions per app. Prior studies have shown that most automated input generation approaches for Android tend to reach near-peak coverage after 5 minutes of exploration [81]. While different input generation approaches tend to exhibit different numbers of actions per given unit of time, our past work shows that our automated input generation approach achieves competitive coverage to similar approaches [35], and our stipulation of 50 actions comfortably exceeds 5 minutes per app. Furthermore, our goal with this large scale analysis was not to completely explore each application, but rather ensure a diverse set of screens and GUI-Component types. For each app the *Execution Engine* extracted `uiautomator` files and screenshot pairs for the top six unique screens of each app based on the number of times the screen was visited. If fewer than six screens were collected for given app, then the information for all screens was collected. Our large scale *Execution Engine* operates in a parallel fashion, where a centralized dispatcher allocated jobs to workers, where each worker is connected to one physical Nexus 7 tablet and is responsible for coordinating the execution of incoming jobs. During the dynamic analysis process, each job consists of the systematic execution of a single app from our dataset. When a worker finished with a job, it then notified the dispatcher which in turn allocates a new job. This process proceeded in parallel across 5 workers until all applications in our dataset had been explored. Since Ads are popular in free apps [82], [83], and are typically made up of dynamic *WebViews* and not native components, we used *Xposed* [84] to block Ads in apps that might otherwise obscure other types of native components.

This process resulted in a dataset of GUI-information and screenshots for 19,786 unique app screens containing over 431,734 native Android GUI-components and containers which, to the best of the authors knowledge, is the largest such dataset of mobile application GUI information ever collected. In Fig. 2 we illustrate the frequency in logarithmic-scale of the top-19 observed components by app category using a heat-map based on the frequency of com-
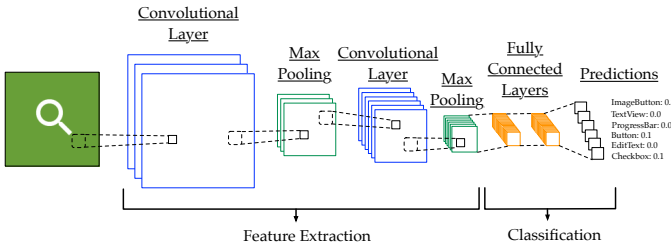
Fig. 3: Typical Components of CNN Architecture

ponents appearing from apps within a particular category (excluding *TextViews* as they are, unsurprisingly, the most popular type of component observed, comprising $\approx 25\%$ of components). The distributions of components in this dataset illustrate two major points. First, while ImageViews and TextViews tend to comprise a large number of the components observed in practice, developers also heavily rely on other types of native Android components to implement key pieces of app functionality. For instance, Buttons, CheckedTextViews, and RadioButtons combined were used over 20k times across the apps in our dataset. Second, we observed certain types of components may be more popular for different categories of apps. For instance, apps from the category of "MUSIC_AND_AUDIO" tend to make much higher use of *SeekBar* and *ToggleButton* components to implement the expected functionalities of a media player, such as scrubbing through music and video files. These findings illustrate that for an approach to be able to effectively generate prototypes for a diverse set of mobile apps, it must be capable of correctly detecting and classifying popular types of GUI-components to support varying functionality.

3.2.3 Phase 2.2 - CNN Classification of GUI-Components

Once the labeled training data set has been collected, we need to train a ML approach to extract salient features from the GUI-component images, and classify incoming images based upon these extracted features. To accomplish this our approach leverages recent advances in CNNs. The main advantage of CNNs over other image classification approaches is that the architecture allows for automated extraction of abstract features from image data, approximation of non-linear relationships, application of the principle of data-locality, and classification in an end-to-end trainable architecture.

For the reminder of this sub-section we give an overview of a typical CNN architecture, explaining elements of the architecture that enable accurate image classification. However, for more comprehensive descriptions of CNNs, we refer readers to [29] & [85]. Fig. 3 illustrates the basic components of a traditional CNN architecture. As with most types of artificial neural networks, CNNs typically encompass several different *layers* starting with an input layer where an image is passed into the network, then to hidden layers where abstract features, and weights representing the "importance" of features for a target task are learned. CNNs derive their name from unique "convolutional" layers which operate upon the mathematical principle of a convolution [86]. The purpose of the convolutional layers, shown in blue in Figure 3, are to extract features from images. Most images are stored as a three (or four) dimensional matrix of numbers, where each dimension of the matrix represents

the intensity of a color channel (*e.g.*, RGB). Convolutional layers operate upon these matrices using a *filter* (also called kernel, or feature detector), which can be thought of as a sliding window of size n by m that slides across an set of matricies representing an image. This window applies a convolution operation (*i.e.*, an element-wise matrix multiplication) creating a *feature map*, which represents extracted image features. As convolution layers are applied in succession, more abstract features are learned from the original image. *Max Pooling* layers also operate as a sliding window, pooling maximum values in the feature maps to reduce dimensionality. Finally, fully-connected layers and a softmax classifier act as a multi-layer perceptron to perform classification. CNN training is typically performed using gradient descent, and back-propagation of error gradients.

The size of the resulting feature map results from three parameters: (i) the number of filters used, (ii) the stride of the sliding window, and (iii) whether or not padding is applied. Leveraging multiple filters allows for multi-dimensional feature maps, the stride corresponds to the distance the sliding window moves during each iteration, and padding can be applied to learn features from the borders of an input image. These feature maps are intended to represent abstract features from images, which inform the prediction process. Traditionally, an element of non-linearity is introduced after each convolutional layer, as the convolution operator is linear in nature, which may not correspond to non-linear nature of data being learned. The typical manner in which this non-linearity is introduced is through *Rectified Linear Units* (ReLUs). The operation these units perform is simple in nature, replacing all negative values in a feature map with zeros. After the convolutions and ReLU operations have been performed, the resulting feature map is typically subjected to *max pooling* (Fig. 3).

Max pooling again operates as as sliding window, but instead of performing a convolution, simply pools the maximum value from each step of the sliding window. This allows for a reduction in the dimensionality of the data while extracting salient features. The layers described thus far in the network have been focused on deriving features from images. Therefore, the final layers of the network must utilize these features to compute predictions about classes for classifications. This is accomplished via the *fully connected layers*, which act as a multi-layer perceptron typically utilizing a softmax activation function.

Training a CNN is accomplished through back-propagation. After the initialization of all the network parameters, initial weights are set to random values. Then input images are fed through the network layers in the forward direction, and the total error across all output classes is calculated. This error is back-propagated through the network and *gradient descent* is used to calculate error gradients for the network weights which are then updated to minimize the output error. A *learning rate* controls the degree to which weights are updated based on the gradient calculations. This process is repeated over the entire training image set, which allows for training both feature extraction and classification in one automated process. After training is complete, the network should be capable of effective classification of input images.

3.2.4 ReDraw Implementation - CNN Classifier

Once the GUI-components in a target mock-up artifact have been detected using either mock-up meta-data or CV-based techniques, REDRAW must effectively classify these components. To accomplish this REDRAW implements a CNN capable of classifying a target image of a GUI-component into one of the 15 most-popular types of components observed in our dataset. In this subsection, we first describe the data-cleaning process used to generate the training, validation, and test datasets (examples of which are shown in Fig. 4) before describing our CNN architecture and the training procedure we employ.

Data Cleaning: We implemented several types of preprocessing and filtering techniques to help reduce noise. More specifically, we implemented filtering processes at three differing levels of granularity: (i) application, (ii) screen & (iii) GUI-component level.

While future versions of REDRAW may support non-native apps, to provide an appropriate scope for rigorous experimentation, we have implemented REDRAW with support for prototpying *native* Android applications. Thus, once we collected the `xml` and screenshot files, it is important to apply filters in order to discard applications that are non-native, including games and hybrid applications. Thus, we applied the following app-level filtering methodologies:

- *Hybrid Applications:* We filtered applications that utilize Apache Cordova [87] to implement mobile apps using web-technologies such as `html` and `CSS`. To accomplish this we first decompiled the `APKs` using `Apktool` [88] to get the resources used in the application. We then discarded the applications that contained a `www` folder with `html` code inside.
- *Non-Standard GUI Frameworks:* Some modern apps utilize third party graphical frameworks or libraries to create highly-customized GUIs. While such frameworks tend to be used heavily for creating mobile games, they can also be used to create UIs for for more traditional applications. One such popular framework is the Unity [89] game engine. Thus, to avoid applications that utilize this engine we filtered out applications that contain the folder structure `com/unity3d/player` inside the code folder after decompilation with `Apktool`.

This resulted in a dataset of 8,655 apps to which we then applied screen-level filtering. At the Screen-level, we implemented the following pre-processing techniques:

- *Filtering out Landscape screens:* To keep the height and width of all screens consistent, we only collected data from screens that displayed in the portrait orientation. Thus, we checked the size of the extracted screenshots and verified that the width and the height correspond to 1200x1920, the screen size used on our target Nexus 7 devices. However, there are some corner cases in which the images had the correct portrait size but it was on landscape. So, to overcome this we checked the extracted `uiautomator xml` file and validated the size of the screen to ensure a portrait orientation.
- *Filtering Screens containing only Layout components:* In Android, Layout components are used as containers that group together other types of functional compo-
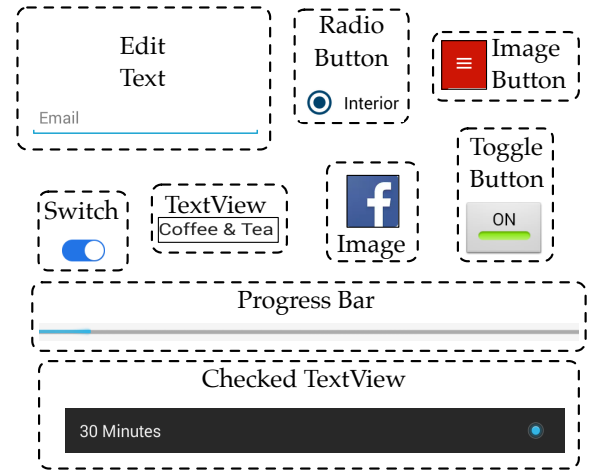

Fig. 4: Example of a subset of ReDraw's training data set consisting of GUI-Component sub-images and domain (Android) specific labels. Images and corresponding Labels are grouped according to the dashed-lines.

nents such as *Buttons* and *Spinners*. However, some screens may only consist only of layout components, thus to ensure variety in our dataset, we analyzed the `uiautomator xml` files extracted during dynamic analysis to discard screens that are only comprised of `Layout` components such as *LinearLayout, GridLayout, and FrameLayout* among others.

- *Filtering WebViews:* While many of the most popular Android apps are native, some apps may be hybrid in nature, that is utilizing web content within a native app wrapper. Because such apps use components that cannot be extracted via `uiautomator` we discard them from our dataset by removing screens where a `WebView` occupied more than 50% of the screen.

After these filtering techniques were applied the resulting dataset contained 14,382 unique screens. We used the information in the `uiautomator xml` files to extract the bounding boxes of *leaf-level* GUI-components in the GUI-hierarchies. We only extract leaf-level components in order to align our dataset with components detected from mock-ups. Intuitively it is unlikely that container components (*e.g.*, non-leaf nodes) would exhibit significant distinguishable features that a ML approach would be able to derive in order to perform accurate classification (hence, the use of our KNN-based approach is described in Sec. 3.3). Furthermore, it is unclear how such a GUI-container classification network would be used to iteratively build a GUI-structure. We performed a final filtering of the extracted leaf components:

- *Filtering Noise:* In our dataset, we observed that in rare cases the bounds of components would not be valid (*e.g.*, extending beyond the borders of the screen, or representing negative areas). Thus, we filter out these cases.
- *Filtering Solid Colors:* We also observed that in certain circumstances, extracted components were made up of a single solid color, or in rarer cases two solid colors. This typically occurred due to instances where the view hierarchy of a screen had loaded, but the content was still rendering on the page or being loaded over the network, when a screenshot was captured. Thus, we discarded such cases.
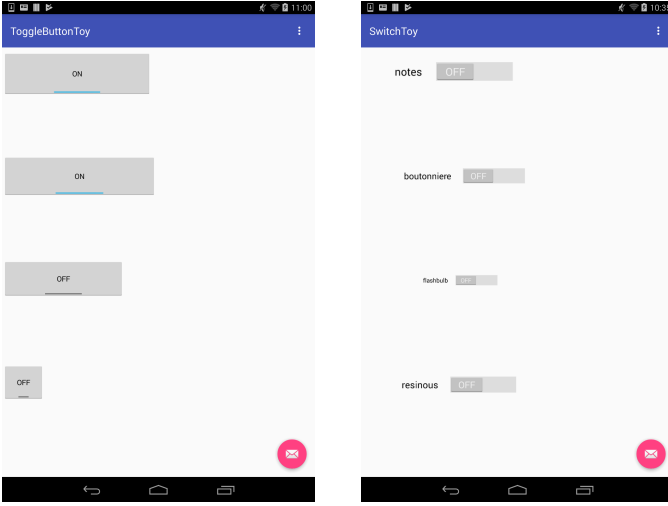
Fig. 5: Screenshots of synthetically generated applications containing toggle buttons and switches

- **Filtering Rare GUI-Components:** In our dataset we found that some components only appeared very few times, therefore, we filtered out any component with less than 200 instances in the initial dataset, leading to 15 GUI-component types in our dataset.

The data-cleaning process described above resulted in 191,300 labeled images of GUI-components. To ensure the integrity of our dataset, we randomly sampled a statistically significant sample of 1,000 GUI-component images (corresponding to confidence interval of ± 3.09 at a 95% confidence level), and had one author manually inspect all 1,000 images and labels to ensure the dataset integrity.

Data Augmentation: Before segmenting the resulting data into training, test, and validation sets, we followed procedures from previous work [29] and applied data augmentation techniques to increase the size of our dataset in order to ensure proper training support for underrepresented classes and help to combat overfitting to the training set. Like many datasets procured using "naturally" occurring data, our dataset suffers from imbalanced classes. That is, the number of labeled images in our training set are skewed toward certain classes, resulting in certain classes that have high support, and others that have low support. Thus, to balance our dataset, we performed two types of data augmentation: *synthetic app generation* and *color perturbation*. For the sake of clarity, we will refer to data collected using our automated dynamic analysis approach as *organic data* (*i.e.*, the data extracted from Google Play) and data generated via synthetic means as *synthetic data* (*i.e.*, generated either via synthetic app generation or color perturbation).

To generate synthetic data for underrepresented components, we implemented an *app synthesizer* capable of generating Android apps consisting of only underrepresented components. The app synthesizer is a Java application that is capable of automatically generating single-screen Android applications containing four instances of GUI-components (with randomized attributes) for 12 GUI-component classes in our dataset that had less than 10K observable instances. The synthesizer places the four GUI-components of the specified type on a single app screen with randomized sizes

and values (*e.g.*, numbers for a number picker, size and state for a toggle button). Two screenshots of synthesized applications used to augment the Toggle button and Switch classes are illustrated in Fig. 5. We ran these apps through our *Execution Engine*, collecting the `uiautomator xml` files and screenshots from the single generated screen for each app. After the screenshots and uiautomator files were collected, we extracted *only* the target underrepresented components from each screenshot (note that in Fig. 5 there is a header title and button generated when creating a standard Android app), all other component types are ignored. 250 apps for each underrepresented GUI-component were synthesized, resulting in creating an extra 1K components for each class and 12K total additional GUI-components.

While our application generator helps to rectify the imbalanced class support to an extent, it does not completely balance our classes and may be prone to overfitting. Thus, to ensure proper support across all classes and to combat overfitting, we follow the guidance outlined in related work [29] to perform *color perturbation* on both the organic and synthetic images in our dataset. More specifically, our color perturbation procedure extracts the RGB values for each pixel in an input image and converts the values to the HSB (Hue, Saturation, Brightness) color space. The HSB color space model represents colors as part of a cylindrical or cone model where color hues are represented by degrees. Thus, to shift the colors of a target image, our perturbation approach randomly chooses a degree value by which each pixel in the image is shifted. This ensures that color hues that were the same in the original image, all shift to the same new color hue in the perturbed image, preserving the visual coherency of the perturbed images. We applied color perturbation to the training set of images until each class of GUI-component had at least 5K labeled instances, as described below.

Data Segmentation: We created a the training, validation, and test datasets for our CNN such that the training dataset contained both *organic* and *synthetic* data, but the test and validation datasets contained **only** *organic* data, unseen in the training phase of the CNN. To accomplish this, we randomly segmented our dataset of *organic* components extracted from Google Play into *training* (75%), *validation* (15%), and *test* (10%) sets. Then for the training set, we added the synthetically generated components to the set of organic GUI-component training images, and performed color perturbation on **only** the training data (after segmentation) until each class had at least 5K training examples. Thus, the training set contained *both* organic and synthetically generated data, and the validation and test sets contained *only* organic data. This segmentation methodology closely follows prior work on CNNs [29].

ReDraw's CNN Architecture: Our CNN architecture is illustrated in Fig. 6. Our network uses an architecture similar to that of AlexNet [29], with two less convolutional layers (3 instead of 5), and is implemented in `MATLAB` using the Neural Network [90], Parallel Computing [91], and Computer Vision [92] toolkits. While "deeper" architectures do exist [30], [32], [33] and have been shown to achieve better performance on large-scale image recognition benchmarks, this

Network Layers

Input Layer

Convolutional Layer + ReLU

Convolutional Layer + ReLU

MaxPooling Layer

Convolutional Layer + ReLU

MaxPooling Layer

Dropout Layer

FullyConnected + ReLU

Dropout Layer

FullyConnected + ReLU

FullyConnected + ReLU

Softmax Layer

Parameters

Filter Size=7, numFilters =64, padding=3, stride=2

Filter Size=7, numFilters =64, padding=3, stride=2

poolSize=3x3, stride=1

Filter Size=3, numFilters =96

poolSize=2x2, stride=1

Dropout Rate = 0.5

1024 nodes

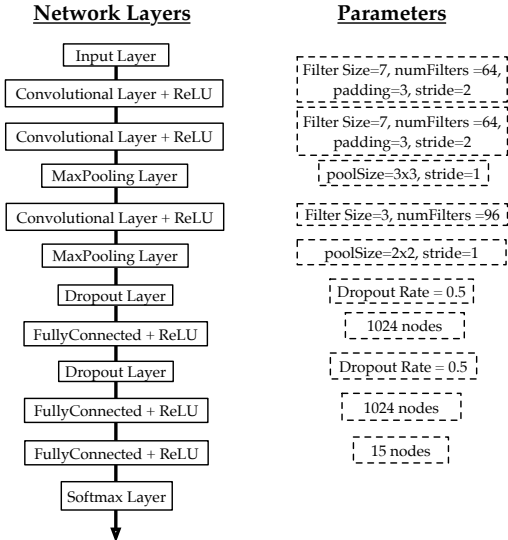Dropout Rate = 0.5

1024 nodes

15 nodes

Fig. 6: REDRAW CNN Architecture

comes at the cost of dramatically longer training times and a larger set of parameters to tune. Since our goal is to classify 15 classes of the most popular Android GUI-components, we do not need the capacity of deeper networks aiming to classify thousands of image categories. We leave deeper architectures and larger numbers of image categories as future work. Also, this allowed our CNN to converge in a matter of hours rather than weeks, and as we illustrate, still achieve high precision.

To tune our CNN, we performed small scale experiments by randomly sampling 1K images from each class to build a small training/validation/test set (75%, 15%, 10%) for faster training times (Note these datasets are separate from the full set used to train/validate/test the network described earlier). During these experiments we iteratively recorded the accuracy on our validation set, and recorded the final accuracy on the test set. We tuned the location of layers and parameters of the network until we achieved peak test accuracy with our randomly sampled dataset.

Training the CNN: To train REDRAW'S network we utilized our derived training set; we trained our CNN end-to-end using back-propagation and stochastic gradient descent with momentum (SGDM), in conjunction with a technique to prevent our network from overfitting to our training data. That is, every five epochs (*e.g.*, entire training set passing through the network once) we test the accuracy of our CNN on the validation set, saving a copy of the learned weights of the classifier at the same time. If we observe our accuracy decrease for more than two checkpoints, we terminate the training procedure. We varied our learning rate from 0.001 to 1×10^{-5} after 50 epochs, and then dropped the rate again to 1×10^{-6} after 75 epochs until training terminated. Gradually decreasing the learning rate allows for the network to "fine-tune" the learned weights over time, leading to an increase in overall classification precision [29]. Our network training time was 17 hours, 12 minutes on a machine with a single Nvidia Tesla K40 GPU.

Using the CNN for Classification: Once the CNN has been trained, new, unseen images can fed into the network

---

Algorithm 1: Hierarchy Construction

Input: InputNodes // classified components from CNN
Output: HierarchyRoot // Root of the constructed GUI-hierarchy

1 **begin**
   // Find containers while there are still components to be
   // processed or until the desired hierarchy depth is reached
2   **while** *(InputNodes.size() > 1) ∧ (count ¡ maxHierarchyDepth)* **do**
3   *Containers* = **containerDetermination** (InputNodes)
4   *count = count +1.*
5   **if** *InputNodes.size() == 1* **then**
6   *RootNode = InputNode* // if 1 node remains
7   // that is the root node
8   **else**
   // else, remaining nodes become children of new root
9   *RootNode* = createNewRootNode()
10   *RootNode*.setChildren(Contaiers)

Algorithm 2: KNN Container Determination

Input: InputNodes // Either leaf components or other containers
Output: Containers // Groupings of input components

1 **while** *canGroupMoreNodes()* // stop if no groupings possible
2 **do**
   // For each container grouping in the mined data
3   **foreach** *D ∈ Data* **do**
4   *score = $\frac{D \cap InputNodes}{D \cup InputNodes}$* // intersection over union
5   **if** *score > curmax* **then**
6   *curmax = score*
7   *Container = D*
8   **end**
9   **end**
10   *Container*.setChildren(*container ∩ I*)
   InputNodes.remove(*container ∩ InputNodes*)
   Containers.addContainer(*Container*)
11 **end**

---

resulting a series of classification scores corresponding to each class. In the case of ReDraw, the component class with the highest confidence is assigned to be the label for a given target image. We present an evaluation of the classification accuracy of REDRAW'S CNN using the dataset described in this subsection later in Sec. 4 & 5.

3.3 Phase 3 - Application Assembly

The final task of the prototyping process is to assemble app GUI code, which involves three phases (Fig. 1-③): (i) building a proper hierarchy of components and containers, (ii) inferring stylistic details from a target mock-up artifact, and (iii) assembling the app.

3.3.1 Deriving GUI-Hierarchies

In order to infer a realistic hierarchy from the classified set of components, our approach utilizes a KNN technique (Alg. 1 & 2) for constructing the GUI hierarchy. This algorithm takes the set of detected and classified GUI-components represented as nodes in a single level tree (*InputNodes*) as input. Starting with all components Alg. 1 starts with the classified leaf nodes represented as output from the CNN and passes them to Alg. 2 which finds component groupings by matching the input leaf components to other sets of leaf nodes observed in GUI-hierarchies collected during automated dynamic analysis using an intersection

Listing 1: ReDraw's Skeleton Main Activity Class

```
1  public class MainActivity extends Activity {
2      @Override
3      protected void onCreate(Bundle
           savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_activity);
6      }
7  }
```

over union (IOU) of screen area occupied similarity metric (Alg. 2 -line 4). Once a matching set of components from the mined data has been found, all *InputNodes* that overlap with the leaf nodes of the matched hierarchy are grouped according to the container observed in the data, and removed from collection of components to match. This process is repeated for the remaining components. This procedure is also applied iteratively (including grouping containers in other containers) until a specified number of levels in the hierarchy are built or all nodes have been grouped. The result of this process is a hierarchy built according to its similarity to existing GUI-hierarchies observed in data. Given different types of containers may behave differently, this technique has the advantage that, in addition to leaf level GUI-components being properly classified by the CNN, proper types of container components are built into the GUI-hierarchy via this KNN-based approach.

3.3.2 Inferring Styles and Assembling a Target App

To infer stylistic details from the mock-up, our approach employs the CV techniques of Color Quantization (CQ), and Color Histogram Analysis (CHA). For GUI-components whose type does not suggest that they are displaying an image, our approach quantizes the color values of each pixel and constructs a color histogram. The most popular color values can then be used to inform style attributes of components when code is generated. For example, for a component displaying text, the most prevalent color can be used as a background and the second most prevalent color can be used for the font.

Target apps are assembled according to their specific domain. However, the process involves translating the derived GUI-hierarchy of an app into the code required to display the hierarchy on a screen. For example, printing formatted `html` for web apps or `xml` layout in Android apps. This is primarily an engineering effort.

3.3.3 ReDraw Implementation - App Assembly

ReDraw assemebles Android applications, using the KNN approach for GUI-hierarchy construction (see Sec. 3.3.1) and CV-based detection of color styles. The input to Alg. 1 is the set of classified "leaf-node" components from the CNN, and the output is a a GUI-hierarchy. To provide sufficient data for the KNN-algorithm, a corpus including all of the info from the "cleaned" screens of the mined GUI-hierarchies mined from our large scale dynamic analysis process is constructed. This corpus forms the dataset which the target set of input leaf node components are matched against during hierarhcy construction. The GUI-hierarchy generated by the KNN for the target "leaf-node" components is then used to infer stylistic details from the original mock-up artifact.

Listing 2: Snippet from `layout.xml` file generated by ReDraw for the Yelp Application

```
1  <LinearLayout android:id="@+id/LinearLayout452"
       android:layout_height="127.80186dp"
       android:layout_marginStart="0.0dp"
       android:layout_marginTop="0.0dp"
       android:layout_width="400.74304dp"
       android:orientation="vertical" android:text=
       "" android:textSize="8pt">
2      <Button android:id="@+id/Button454"
         android:layout_height="58.45201dp"
         android:layout_marginStart="0.0dp"
         android:layout_marginTop="0.0dp"
         android:layout_width="400.74304dp"
         android:text="Sign up with Google"
         android:textSize="8pt" style="@style/
         Style65"/>
3      <Button android:id="@+id/Button453"
         android:layout_height="50.526318dp"
         android:layout_marginStart="3.4674923dp"
          android:layout_marginTop="18.82353dp"
         android:layout_width="393.31268dp"
         android:text="Sign up with Facebook"
         android:textSize="8pt" style="@style/
         Style66"/>
4  </LinearLayout>
```

Listing 3: Snippet from `style.xml` file generated by ReDraw for the Yelp Application

```
1  <style name="Style63" parent="AppTheme">
2      <item name="android:textColor">#FEFEFF</item>
3  </style>
4  <style name="Style64" parent="AppTheme">
5      <item name="android:textColor">#FEFEFF</item>
6  </style>
7  <style name="Style65" parent="AppTheme">
8      <item name="android:background">#DD4B39</item
         >
9      <item name="android:textColor">#FEFEFF</item>
10 </style>
```

More specifically, for each component and container, we perform CQ and CHA to extract the dominant colors for each component. For containers, the background color is set to the dominant color, for text components, the dominant color is used as the background and the second-most dominant color is used as the font-color. For components which have a text element, we apply optical character recognition (OCR) using the open source Tesseract [93] library on the original screenshot to obtain the text strings.

REDRAW encodes the information regarding the GUI-hierarchy, stylistic details, and strings detected using OCR into an intermediate representation (IR) before translating it into code. This IR follows the format of `uiautomator` `xml` files that describes dynamic information from an Android screen. Thus, after REDRAW encodes the GUI information into the `uiautomator`-based IR, it then generates the necessary resource `xml` files (*e.g.*, files in the `res` folder of an Android app project directory) by parsing the `uiautomator`-based IR `xml` file. This process generates the following two types of resource code for the generated app: (i) the `layout.xml` code describing the general GUI structure complete with strings detected via OCR; and (ii) a `style.xml` file that stipulates the color and style information for each component gleaned via the CV techniques, and ReDraw generates the xml source files following the best practices stipulated in the Android developer guidelines [39], such as utilizing relative positioning, and proper

padding and margins. In addition to these resource `xml` files REDRAW also generates a skeleton Java class encompassing the `MainActivity` which renders the GUI stipulated in the resource `xml` files, as well as other various files required to build and package the code into an `apk`. The Skeleton `MainActivity` Java class is shown in Listing 1 and snippets from generated `layout.xml` & `style.xml` files for a screen from the Yelp application are shown in Listings 2 & 3. The `layout.xml` snippet of code generated by ReDraw illustrates the use of margins and relative `dp` values to stipulate the spatial properties of GUI-containers and GUI-components and references the `style.xml` file to stipulate color information. Listing 3 illustrates the corresponding styles and colors referenced by the `layout.xml` file.

4 EMPIRICAL STUDY DESIGN

The *goal* of our empirical study is to evaluate REDRAW in terms of (i) the accuracy of the CNN GUI-component classifier, (ii) the similarity of the generated GUI-hierarchies to real hierarchies constructed by developers, (iii) the visual similarity of generated apps compared to mock-ups, and (iv) ReDraw's suitability in an industrial context. The *context* of this study consists of (i) a set of 191,300 labeled images of Android GUI-components extracted from 14,382 unique app screens mined from 8,655 `APKs` mined from the Google Play store (top-240 rated apps in each category excluding games and duplicates, see Sec. 3.2.2) to assess the accuracy of the CNN-classifier, (ii) 83 additional screens (not included in the dataset to train and test the CNN-classifier) extracted from 32 of the highest rated apps on Google Play (top-3 in each category), (iii) nine reverse engineered Sketch mockups from eight randomly selected highly rated Google Play Apps to serve as mock-up artifacts, and (iv) two additional approaches for prototyping Android applications REMAUI [8] and pix2code [37]. The *quality focus* of this study is the effectiveness of REDRAW to generate prototype apps that are both visually similar to target mock-up artifacts, with GUI-hierarchies similar to those created by developers. To aid in achieving the goals of our study we formulated the following RQs:

- *RQ$_1$: How accurate is the CNN-based image classification for classifying Android GUI-components?*
- *RQ$_2$: How similar are GUI-hierarchies constructed using REDRAW'S KNN algorithm compared to real GUI-hierarchies?*
- *RQ$_3$: Are the prototype applications that REDRAW generates visually similar to mock-up artifacts?*
- *RQ$_4$: Would actual mobile developers and designers consider using REDRAW as part of their workflow?*

4.1 RQ$_1$: Effectiveness of the CNN

To answer **RQ$_1$**, as outlined in Sec. 3.2.4 we applied a large scale automated dynamic analysis technique and various data cleaning procedures which resulted in a total of 8,655 apps, 14,382 unique screens, and 191,300 labeled images of GUI-components. To normalize support across classes and prepare training, validation and test sets in order measure the effectiveness of our CNN we applied data augmentation, and segmentation techniques also described in detail in Sec. 3.2.4. The datasets utilized are illustrated, broken down by class, in Table 1. We trained the CNN on the training

TABLE 1: Labeled GUI-Component Image Datasets

GUI-C Type	Total # (C)	Tr (O)	Tr (O+S)	Valid	Test
TextView	99,200	74,087	74,087	15,236	9,877
ImageView	53,324	39,983	39,983	7,996	5,345
Button	16,007	12,007	12,007	2,400	1,600
ImageButton	8,693	6,521	6,521	1,306	866
EditText	5,643	4,230	5,000	846	567
CheckedTextView	3,424	2,582	5,000	505	337
CheckBox	1,650	1,238	5,000	247	165
RadioButton	1,293	970	5,000	194	129
ProgressBar	406	307	5,000	60	39
SeekBar	405	304	5,000	61	40
NumberPicker	378	283	5,000	57	38
Switch	373	280	5,000	56	37
ToggleButton	265	199	5,000	40	26
RatingBar	219	164	5,000	33	22
Spinner	20	15	5,000	3	2
Total	191,300	143,170	187,598	29,040	19,090

Abbreviations for column headings: "Total#(C)"=Total # of GUI-components in each class after cleaning; "Valid"= Validation; "Tr(O)"= Training Data (Organic Components Only); "Tr(O+S)"= Training Data (Organic + Synthetic Components).

set of data, avoiding overfitting using a validation set as described in Sec. 3.2.4. To reiterate, all of the images in the test and validation sets were extracted from real applications and were separate (*e.g.*, unseen) from the training set. To evaluate the effectiveness of our approach we measure the average top-1 classification precision across all classes on the Test set of data:

$$P = \frac{TP}{TP + FP}$$

where TP corresponds to true positives, or instances where the top class predicted by the network is correct, and FP corresponds to false positives, or instances where the top classification prediction of the network is not correct. To illustrate the classification capabilities of our CNN, we present a confusion matrix with precision across classes in Sec. 5. The confusion matrix illustrates correct true positives across the highlighted diagonal, and false positives in the other cells. To help justify the need and applicability of a CNN-based approach, we measure the classification performance of our CNN against a baseline technique, as recent work has suggested that deep learning techniques applied to SE tasks should be compared to simpler, less computationally expensive alternatives [94]. To this end, we implemented a baseline Support Vector Machine (SVM) for classification based image classification approach [95] that utilizes a "Bag of Visual Words" (BOVW). At a high level, this approach extracts image features using the Speeded-Up Robust Feature (SURF) detection algorithm [96], then uses K-means clustering to cluster similar features together, and utilizes an SVM trained on resulting feature clusters. We utilized the same training/validation/test set of data used to the train the CNN and followed the methodology in [95] to vary the number of K-means clusters from $k = 1,000$ to $k = 5,000$ in steps of 50, finding that $k = 4,250$ achieved the best performance in terms of classification precision for our dataset. We also report the confusion matrix of precision values for this technique utilizing the same formulation of the precision metric stated earlier.

4.2 RQ$_2$: GUI Hierarchy Construction

In order to answer **RQ$_2$** we aim to measure the similarity of the GUI-hierarchies in apps generated by REDRAW compared to a ground truth set of hierarchies and a set

of hierarchies generated by two baseline mobile app prototyping approaches, REMAUI and pix2code. To carry out this portion of the study, we selected 32 apps from our cleaned dataset of `Apks` by randomly selecting one of the top-10 apps from each category (grouping all "Family" categories together). We then manually extracted 2-3 screenshots and `uiautomator xml` files per app, which were not included in the original dataset used to train, validate or test the CNN. After discarding screens according to our filtering techniques, this resulted in a set of 83 screens. Each of these screens was used as input to REDRAW, REMAUI, and pix2code from which a prototype application was generated. Ideally, a comparison would compare the GUI-related source code of applications (*e.g.*, `xml` files located in the `res` folder of Android project) generated using various automated techniques however, the source code of many of the subject Google Play applications is not available. Therefore, to compare GUI-hierarchies, we compare the runtime GUI-hierarchies extracted dynamically from the generated prototype apps for each approach using `uiautomator` to the set of "ground truth" `uiautomator xml` files extracted from the original applications. The `uiautomator` representation of the GUI is a reflection of the constructed GUI-related source code displayed at runtime on the device screen, allowing us to make an accurate comparison of the hierarchal representation of GUI-components and GUI-containers for each approach.

To provide a performance comparison to REDRAW, we selected the two most closely related approaches in related research literature, REMAUI [8] and pix2code [37] , to provide a comparative baseline. To provide a comparison against pix2code, we utilized the code provided by the authors of the paper on GitHub [97] and the provided training dataset of synthesized applications. We were not able to train the pix2code approach on our mined dataset of Android application screenshots for two reasons: (i) pix2code uses a proprietary domain specific language (DSL) that training examples must be translated to and the authors do not provide transformation code or specifications for the DSL, (ii) the pix2code approach requires the GUI-related source code of the applications for training, which would have needed to be reverse engineered from the Android apps in our dataset from Google Play. To provide a comparison against REMAUI [8], we re-implemented the approach based on the details provided in the paper, as the tool was not available as of the time of writing this paper[3].

As stated in Sec. 3.1 REDRAW enables two different methodologies for for *detecting* GUI-components from a mock-up artifact: (i) CV-based techniques and (ii) parsing information directly from mock-up artifacts. We consider both of these variants in our evaluation which we will refer to as REDRAW-CV (for the CV-based approach) and REDRAW-Mockup (for the approach that parses mock-up metadata). Our set of 83 applications from Google Play does not contain traditional mock-up artifacts that would arise as part of the app design and development process (*e.g.*, Photoshop or Sketch files) and reverse engineering these artifacts is an extremely time-consuming task (see

TABLE 2: Semi-Structured Interview Questions for Developers & Designers

Q#	Question Text
Q1	Given the scenario where you are creating a new user interface, would you consider adopting ReDraw in your design or development workflow? Please elaborate.
Q2	What do you think of the visual similarity of the Re-Draw applications compared to the original applications? Please elaborate.
Q3	Do you think that the GUI-hierarchies (*e.g.*, groupings of components) generated by ReDraw are effective? Please elaborate.
Q4	What improvements to ReDraw would further aid the mobile application prototyping process at your company? Please elaborate.

Sec. 4.4). Thus, because manually reverse-engineering mock-ups from 83 screens is not practical, REDRAW-Mockup was modified to parse *only* the bounding-box information of leaf node GUI-components from `uiautomator` files as a substitute for mock-up metadata.

We compared the runtime hierarchies of all generated apps to the original, ground truth runtime hierarchies (extracted from the original `uiautomator xml` files) by deconstructing the trees using pre-order and using the Wagner-Fischer [99] implementation of Levenshtein edit distance for calculating similarity between the hierarchical (*i.e.*, tree) representations of the runtime GUIs. The hierarchies were deconstructed such that the *type* and *nested order* of components are included in the hierarchy deconstruction. We implemented the pre-order traversal in this way to avoid small deviations in other attributes included in the `uiautomator` information, such as pixel values, given that the main *goal* of this evaluation is to measure hierarchical similarities.

In our measurement of edit distance, we consider three different types of traditional edit operations: insertion, deletion, and substitution. In order to more completely measure the similarity of the prototype app hierarchies to the ground truth hierarchies, we introduced a weighting schema representing a "penalty" for each type of edit operation, wherein the default case each operation carries an identical weight of $1/3$. We vary the weights of each edit and calculate a distribution of edit distances which are dependent on the fraction of the total penalty that a given operation (*i.e.*, insertion, deletion, or substitution) occupies, and carry out these calculations varying each operation separately. The operations that are not under examination split the difference of the remaining weight of the total penalty equally. For example, when insertions are given a penalty of 0.5, the penalties for deletion and substitution are set to 0.25 each. This helps to better visualize the minimum edit distance required to transform a REDRAW, or REMAUI generated hierarchy to the original hierarchy and also helps to to better describe the nature of the inaccuracies of the hierarchies generated by each method.

4.3   RQ₃: Visual Similarity

One of REDRAW'S goals is to generate apps that are visually similar to target mock-ups. Thus to answer RQ₃, we compared the visual similarity of apps generated by REDRAW, and REMAUI, using the same set of 83 apps from **RQ₂**. The subjects of comparison for this section of the study were screenshots collected from the prototype

---

3. REMAUI is partially available as a web-service [98], but it did not work reliably and we could not generate apps using this interface.

TABLE 3: Confusion Matrix for REDRAW

	Total	TV	IV	Bt	S	ET	IBt	CTV	PB	RB	TB	CB	Sp	SB	NP	RBt
TV	9877	94%	3%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IV	5345	5%	93%	1%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Bt	1600	11%	6%	81%	0%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	37	5%	3%	0%	87%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
ET	567	14%	3%	2%	0%	81%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IBt	866	4%	23%	1%	0%	0%	72%	0%	0%	0%	0%	0%	0%	0%	0%	0%
CTV	337	7%	0%	0%	0%	0%	0%	93%	0%	0%	0%	0%	0%	0%	0%	0%
PB	41	15%	29%	0%	0%	0%	0%	0%	56%	0%	0%	0%	0%	0%	0%	0%
RB	22	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%
TBt	26	19%	22%	7%	0%	0%	0%	0%	0%	0%	52%	0%	0%	0%	0%	0%
CB	165	12%	7%	0%	0%	1%	0%	0%	0%	0%	0%	81%	0%	0%	0%	0%
Sp	2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
SB	39	10%	13%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	78%	0%	0%
NP	40	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	95%	0%
RBt	129	4%	3%	2%	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	89%

TABLE 4: Confusion Matrix for BOVW Baseline

	Total	TV	IV	Bt	S	ET	IBt	CTV	PB	RB	TB	CB	Sp	SB	NP	RBt
TV	9877	59%	4%	9%	1%	6%	2%	8%	6%	0%	1%	2%	0%	1%	0%	2%
IV	5345	4%	51%	4%	1%	2%	11%	2%	18%	1%	1%	3%	0%	2%	0%	2%
Bt	1600	6%	6%	59%	1%	5%	4%	7%	4%	0%	1%	1%	0%	0%	3%	1%
S	37	5%	0%	3%	65%	0%	0%	5%	22%	0%	0%	0%	0%	0%	0%	0%
ET	567	6%	2%	4%	1%	62%	1%	4%	15%	0%	0%	1%	0%	0%	4%	1%
IBt	866	2%	16%	3%	0%	2%	61%	1%	9%	1%	1%	2%	0%	2%	0%	3%
CTV	337	3%	1%	7%	1%	3%	0%	81%	1%	0%	0%	2%	0%	0%	0%	2%
PB	41	0%	24%	2%	0%	2%	5%	2%	54%	0%	0%	2%	2%	2%	0%	2%
RB	22	0%	5%	0%	0%	0%	0%	0%	27%	68%	0%	0%	0%	0%	0%	0%
TBt	26	7%	7%	19%	0%	0%	0%	11%	15%	0%	33%	0%	0%	0%	0%	7%
CB	165	4%	2%	3%	1%	2%	1%	2%	12%	1%	0%	72%	0%	0%	0%	1%
Sp	2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
SB	39	0%	5%	0%	0%	0%	0%	0%	18%	3%	0%	5%	0%	68%	0%	3%
NP	40	3%	0%	5%	0%	3%	0%	5%	0%	0%	0%	0%	0%	0%	84%	0%
RBt	129	6%	3%	5%	1%	3%	0%	6%	18%	0%	1%	1%	0%	1%	0%	55%

Abbreviations for column headings representing GUI-component types: TextView (TV), ImageView (IV), Button (Bt), Switch (S), EditText (ET), ImageButton (IBt), CheckedTextView (CTV), ProgressBar (PB), RadioButton (RB), ToggleButton (TBt), CheckBox (CB), Spinner (Sp), SeekBar (SB), NumberPicker (NP), RadioButton (RBt)

applications generated by REDRAW-CV, REDRAW-Mockup, pix2code, and REMAUI. Following the experimental settings used to validate REMAUI [8], we used the open source PhotoHawk [100] library to measure the *mean squared error* (MSE) and *mean average error* (MAE) of screenshots from the generated prototype apps from each approach compared to the original app screenshots. To examine whether the MAE and MSE varied to a statistically significant degree between approaches, we compare the MAE & MSE distributions for each possible pair of approaches using a two-tailed Mann-Whitney test [101] (p-value). Results are declared as statistically significant at a 0.05 significance level. We also estimate the magnitude of the observed differences using the Cliff's Delta (d), which allows for a nonparametric effect size measure for ordinal data [102].

4.4 RQ$_4$: Industrial Applicability

Ultimately, the goal of REDRAW is integration into real application development workflows, thus as part of our evaluation, we aim to investigate REDRAW's applicability in such contexts. To investigate **RQ$_4$** we conducted semi-structured interviews with a front-end Android developer at Google, an Android UI designer from Huawei, and a mobile researcher from Facebook. For each of these three participants, we randomly selected nine screens from the set of apps used in **RQ$_2$-RQ$_3$** and manually reversed engineered Sketch mock-ups of these apps. We verified the visual fidelity of these mock-ups using the GVT tool [10], which has been used in prior work to detect presentation

failures, ensuring that there were no reported design violations reported in the reverse-engineered mockups. This process of reverse-engineering the mock-ups was extremely time-consuming to reach acceptable levels, with well over ten hours invested into each of the nine mock-ups. We then used REDRAW to generate apps using both CV-based detection and utilizing data from the mock-ups. Before the interviews, we sent participants a package containing the ReDraw generated apps, complete with screenshots and source code, and the original app screenshots and Sketch mock-ups. We then asked a series of questions (delineated in Table 2) related to (i) the potential applicability of the tool in their design/development workflows, (ii) aspects of the tool they appreciated, and (iii) areas for improvement. Our investigation into this research question is meant to provide insight into the applicability of REDRAW to fit into real design development workflows, however, we leave full-scale user studies and trials as future work with industrial collaborators. This study is not meant to be comparative, but rather to help gauge REDRAW'S industrial applicability.

5 EXPERIMENTAL RESULTS

5.1 RQ$_1$ Results: Effectiveness of the CNN

The confusion matrices illustrating the classification precision across the 15 Android component classes for both the CNN-classifier and the Baseline BOVW approach are shown in Tables 3 & 4 respectively. The first column of the matrices illustrate the number of components in the test set, and the

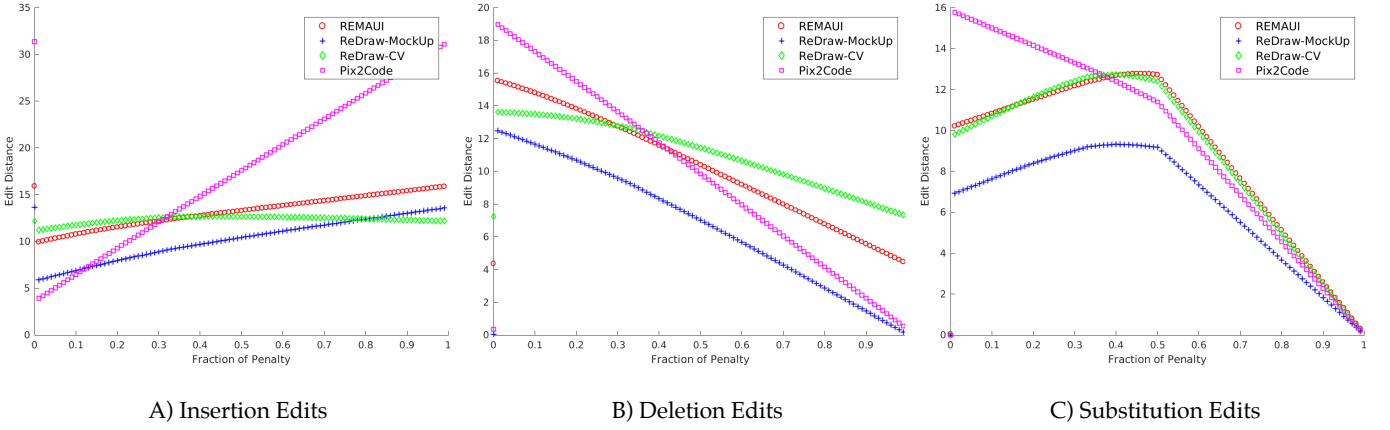| A) Insertion Edits | B) Deletion Edits | C) Substitution Edits |

Fig. 7: Hierarchy similarities based on edit distances

numbers in the matrix correspond to the percentage of each class on the y-axis, that were classified as components on the x-axis. Thus, the diagonal of the matrices (highlighted in blue) corresponds to correct classifications. The overall top-1 precision for the CNN (based on raw numbers of components classified) is 91.1%, whereas for the BOVW approach the overall top-1 precision is 64.7%. Hence, it is clear that the CNN-based classifier that REDRAW employs outperforms the baseline, illustrating the advantage of the CNN architecture compared to a heuristic-based feature extraction approach. In fact, REDRAW'S CNN outperforms the baseline in classification precision *across all classes*.

It should be noted that REDRAW'S classification precision does suffer for certain classes, namely ProgressBars and ToggleButtons. We found that the classification accuracy of these component types was hindered due to multiple existing styles of the components. For instance, the ProgressBar had two primary styles, traditional progress bars, which are short in the y-direction and long in the x-direction, and square progress bars that rendered a progress wheel. With two very distinct shapes, it was difficult for our CNN to distinguish between the drastically different images and learn a coherent set of features to differentiate the two. While the CNN may occasionally misclassify components, the confusion matrix illustrates that these misclassifications are typically skewed toward *similar* classes. For example, ImageButtons are primarily misclassified as ImageViews, & EditTexts are misclassified as TextViews. Such misclassifications in the GUI-hierarchy would be trivial for experienced Android developers to fix in the generated app while the GUI-hierarchy and boilerplate code would be automatically generated by ReDraw. The strong performance of the CNN-based classifier provides a solid base for the application generation procedure employed by ReDraw. Based on these results, we answer **RQ₁:**

RQ$_1$: ReDraw's CNN-based GUI-component classifier was able to achieve a high average precision (91%) and outperform the baseline BOVW approach's average precision (65%).

5.2 RQ$_2$ Results: Hierarchy Construction

An important part of the app generation process is the automated construction of a GUI-hierarchy to allow for

the proper grouping, and thus proper displaying, of GUI-components into GUI-containers. Our evaluation of Re-Draw's GUI-hierarchy construction compares against the REMAUI and pix2code approaches by decomposing the run-time GUI-hierarchies into trees and measuring the edit distance between the generated trees (as described in Section 4.2). By varying the penalty prescribed to each edit operation, we can gain a more comprehensive understanding of the similarity of the generated GUI-hierarchies by observing, for instance, whether certain hierarchies were more or less shallow than real applications, by examining the performance of insertion and deletion edits.

The results for our comparison based on Tree edit distance are illustrated in Fig. 7 A-C. Each graph illustrates the results for a different edit operation and the lines delineated by differing colors and shapes represent the studied approaches (REDRAW Mock-Up or CV-based, REMAUI, or pix2code) with the edit distance (*e.g.*, closeness to the target hierarchy) shown on the y-axis and the penalty prescribed to the edit operation on the x-axis. For each of the graphs, a lower point or line indicates that a given approach was closer to the target mock-up hierarchy. The results indicate that in general, across all three variations in edit distance penalties, REDRAW-MockUp produces hierarchies that are closer to the target hierarchies than REMAUI and pix2code. Of particular note is that as the cost of insertion operations rises both REDRAW-CV and REDRAW-MockUp outperform REMAUI. In general REDRAW-Mockup requires fewer than ten edit operations across the three different types of operations to exactly match the target app's GUI-hierarchy. While REDRAW'S hierarchies require a few edit operations to exactly match the target, this may be acceptable in practice, as there may be more than one variation of an acceptable hierarchy. Nevertheless, REDRAW-Mockup is closer than other related approach in terms of similarity to real GUI-hierarchies.

Another observable phenomena exhibited by this data is the tendency for REMAUI and pix2code to generate relatively *shallow* hierarchies. We see that as the penalty for insertion increases, both REDRAW-CV and REDRAW-Mockup outperform REMAUI and pix2code. This is because ReDraw simply does not have to perform as many insertions into the hierarchy to match the ground truth. Pix2code and REMAUI are forced to add more inner nodes to the tree
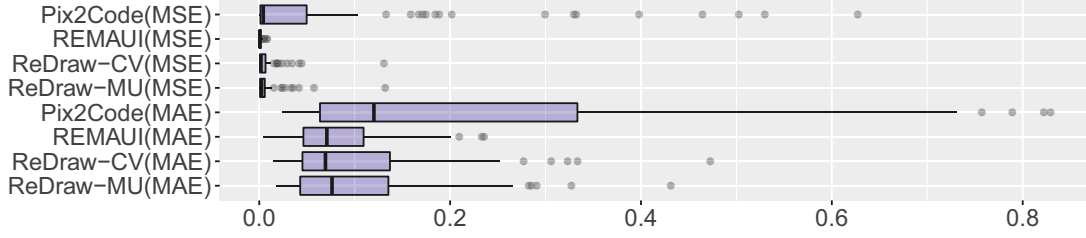
Fig. 8: Pixel-based mean average error and mean squared error of screenshots: REDRAW, REMAUI, and pix2code apps

TABLE 5: Pixel-based comparison by MAE: Mann-Whitney test (p-value) and Cliff's Delta (d).

Test	p-value	d
ReDrawMU *vs* **ReDrawCV**	0.835	0.02 (Small)
ReDrawMU *vs* **REMAUI**	0.542	0.06 (Small)
ReDrawMU *vs* **pix2Code**	< 0.0002	-0.34 (Medium)
pix2Code *vs* **ReDrawCV**	< 0.0001	0.35 (Medium)
pix2Code *vs* **REMAUI**	< 0.0001	0.39 (Medium)
REMAUI *vs* **ReDrawCV**	0.687	-0.04 (Small)

TABLE 6: Pixel-based comparison by MSE: Mann-Whitney test (p-value) and Cliff's Delta (d).

Test	p-value	d
ReDrawMU *vs* **ReDrawCV**	0.771	0.03 (Small)
ReDrawMU *vs* **REMAUI**	< 0.0001	0.45 (Medium)
ReDrawMU *vs* **pix2Code**	< 0.003	-0.27 (Small)
pix2Code *vs* **ReDrawCV**	< 0.002	0.28 (Small)
pix2Code *vs* **REMAUI**	< 0.0001	0.61 (Large)
REMAUI *vs* **ReDrawCV**	<0.0001	-0.42 (Medium)

because their generated hierarchies are too shallow (i.e. lacking in inner nodes). From a development prototyping point of view, it is more likely easier for a developer to remove redundant nodes than it is to create new nodes, requiring them reasoning what amounts to a new hierarchy after the automated prototyping process. These results are unsurprising for the REMAUI approach, as the authors used shallowness as a proxy for suitable hierarchy construction. However, this evaluation illustrates that the shallow hierarchies generated by REMAUI and pix2code do match the target hierarchies as well as those generated by REDRAW-Mockup. While minimal hierarchies are desirable from the point of view of rendering content on the screen, we find that REMAUI's hierarchies tend to be dramatically more shallow compared to REDRAW'S which exhibit higher similarity to real hierarchies. Another important observation is that the substitution graph illustrates the general advantage that the CNN-classifier affords during hierarchy construction. REDRAW-Mockup requires far fewer substitution operations to match a given target hierarchy than REMAUI, which is at least in part due to REDRAW'S ability to properly classify GUI-components, compared to the text/image binary classification afforded by REMAUI. From these results, we can answer **RQ$_2$**:

> **RQ$_2$:** REDRAW-**MockUp is capable of generating GUI-hierarchies closer in similarity to real hierarchies than** REMAUI **or pix2code, and these hierarchies are typically less than ten edits away from target hierarchies for a given edit operation.**

5.3 RQ$_3$ Results: Visual Similarity

An effective GUI-prototyping approach should be capable of generating apps that are visually similar to the target mock-up artifacts. We measured this by calculating the MAE and MSE across all pixels in screenshots for ReDraw-MockUp, REDRAW-CV, REMAUI, and pix2code (Fig. 8.) compared to the original app screenshots. This figure depicts a box-and-whisker plot with points corresponding to a

measurement for each of the studied 83 subject applications. The black bars indicate mean values. In general, the results indicate that all approaches generated apps that exhibited high overall pixel-based similarity to the target screenshots. REDRAW-CV outperformed both REMAUI and pix2code in MAE, whereas all approaches exhibited very low MSE. The apps generated by pix2code exhibit a rather large variation from the target screenshots used as input. This is mainly due to the artificial nature of the training set utilized by pix2code which in turn generates apps only with a relatively rigid, pre-defined set of components. The results of the Mann-Whitney test reported in Table 5 & 6 illustrate wether the similarity between each combination of approaches was statistically significant. For MAE, we see that when REDRAW-CV and REDRAW-Mockup are compared to REMAUI, the results are not statistically significant, however, when examining the MSE for these same approaches the result is statistically significant with a medium effect effect size according to the Cliff's delta measurement. Thus, it is clear that on average REDRAW and REMAUI both generate prototype applications that are closely similar to a target visually, with REDRAW outperforming REMAUI in terms of MSE. This is encouraging, given that REMAUI simply copies images of all component types into its mock-ups rather than generating code with proper component types. When comparing both variants of REDRAW and REMAUI to pix2code, the results are all statistically significant, with ranging effect sizes. Thus, both REDRAW and REMAUI outperform pix2code in terms of generating prototypes that are visually similar to a target.

While in general the visual similarity for apps generated by REDRAW is high, there are instances where REMAUI outperformed our approach. Typically this is due to instances where REDRAW misclassifies a small number of components that cause visual differences. For example, a button may be classified and rendered as a switch in rare cases. However, REMAUI does not suffer from this issue as all components deemed not to be text are copied to the generated app as an image. While this occasionally leads to more visually similar apps, the utility is dubious at best, as developers will be
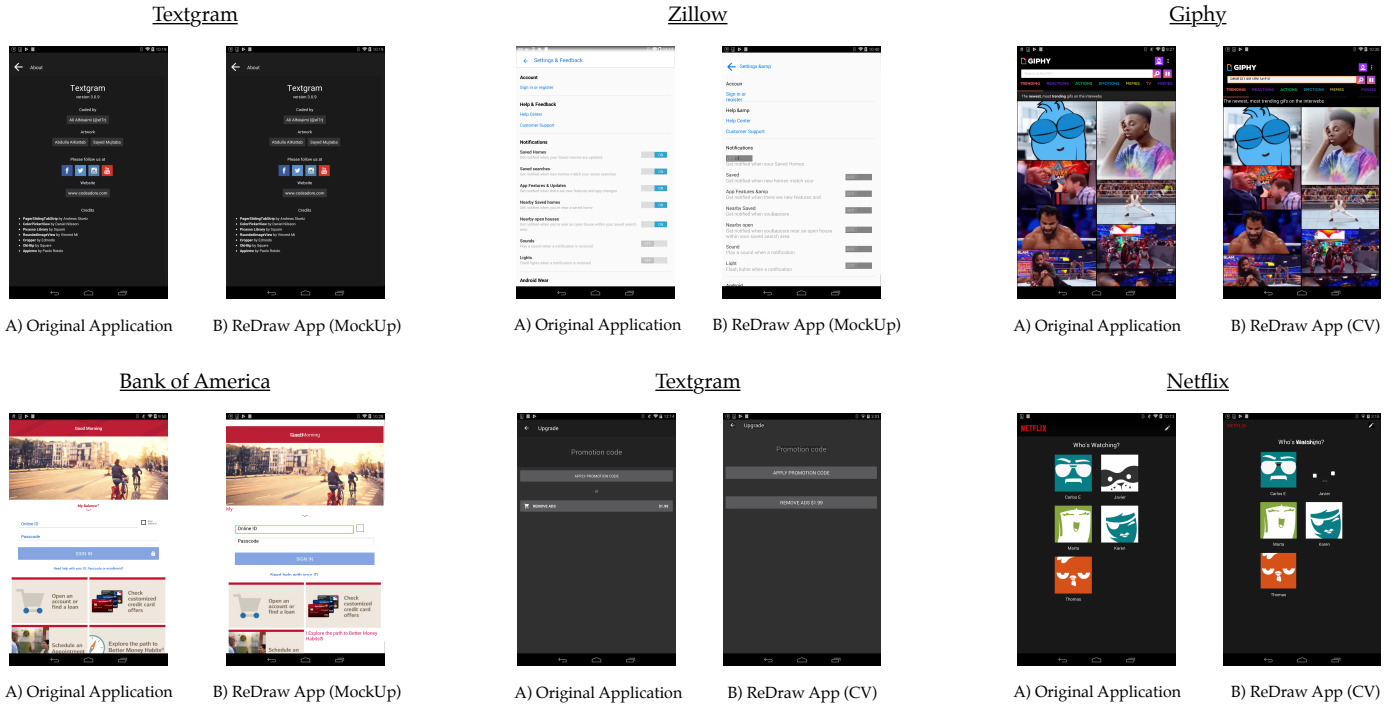
Textgram

A) Original Application   B) ReDraw App (MockUp)

Zillow

A) Original Application   B) ReDraw App (MockUp)

Giphy

A) Original Application   B) ReDraw App (CV)

Bank of America

A) Original Application   B) ReDraw App (MockUp)

Textgram

A) Original Application   B) ReDraw App (CV)

Netflix

A) Original Application   B) ReDraw App (CV)

Fig. 9: Examples of apps generated with REDRAW exhibiting high visual and structural similarity to target apps

required to add proper component types, making extensive edits to the GUI-code. Another instance that caused some visual inconsistencies for REDRAW was text overlaid on top of images. In many cases, a developer might overlay a snippet of text over an image to create a striking effect (*e.g.*, Netflix often overlays text across movie-related images). However, this can cause an issue for REDRAW'S prototyping methodology. During the detection process, REDRAW recognizes images and overlaid text in a mockup. However, given the constraints of our evaluation, REDRAW simply reuses the images contained within screenshot as is, which might include overlaid text. Then, ReDraw would render a `TextView` or `EditText` over the image *which already includes the overlaid text* causing duplicate lines of text to be displayed. In a real-world prototyping scenario, such issues can be mitigated by designers providing "clean" versions of the images used in a mockup, so that they could be utilized in place of "runtime" images that may have overlaid text. Overall, the performance of REDRAW is quite promising in terms of the visual fidelity of the prototype apps generated, with the potential for improvement if adopted into real workflows.

We illustrate some of the more successful generated apps (in terms of visual similarity to a target screenshot) in Fig. 9; screenshots and hierarchies for all generated apps will be available in a dataset in our online appendix [38]. In summary, we can answer **RQ$_3$** as follows:

> **RQ$_3$: The apps generated by ReDraw exhibit high visual similarity compared to target screenshots.**

5.4 RQ$_4$ Results: Industrial Applicability

To understand the applicability of REDRAW from an industrial prospective we conducted a set of semi-structured interviews with a front-end Android developer @Google, a mobile designer @Huawei, and a mobile researcher @Face-

book. We asked them four questions (see Sec. 4) related to (i) the applicability of REDRAW, (ii) aspects of REDRAW they found beneficial, and (iii) areas for improvement.

5.4.1 Front End Android Developer @Google

The first individual works mostly on Google's search products, and his team practices the process of mock-up driven development, where developers work in tandem with a dedicated UI/UX team. Overall, the developer was quite positive about REDRAW explaining that it could help to improve the process of writing a new Android app activity from scratch, however, he noted that *"It's a good starting point... From a development standpoint, the thing I would appreciate most is getting a lot of the boilerplate code done [automatically]"*. In the "boilerplate" code statement, the developer was referring to the large amount of layout and style code that must be created when creating a new activity or view. He also admitted that this code is typically written by hand stating, *"I write all my GUI-code in xml, I don't use the Android Studio editor, very few people use it"*. He also explained that this GUI-code is time-consuming to write and debug stating, *"If you are trying to create a new activity with all its components, this can take hours"*, in addition to the time required for the UI/UX team to verify proper implementation. The developer did state that some GUI-hierarchies he examined tended to have redundant containers, but that these can be easily fixed stating, *"There are going to be edge cases for different layouts, but these are easily fixed after the fact"*.

The aspect of REDRAW that this developer saw the greatest potential for, is its use in an evolutionary context. During the development cycle at Google, the UI/UX team will often propose changes to existing apps, whose GUI-code must be updated accordingly. The developer stated that REDRAW had the potential to aid this process: *"The key thing is fast iteration. A developer could generate the initial view*

[using ReDraw], clean up the layouts, and have a working app. If a designer could upload a screenshot, and without any other intervention [ReDraw] could update the [existing] xml this would be ideal." The developer thought that if REDRAW was able to detect existing GUI-components in a prior app version, and update the layouts and styles of these components according to a screenshot, generating new components as necessary, this could greatly improve the turn around time of GUI-changes and potentially increase quality. He even expressed optimism that the approach could learn from developer corrections on generated code over time, stating *"It would be great if you could give it [ReDraw] developer fixes to the automatically generated xml and it could learn from this."*

5.4.2 Mobile UI/UX Designer @Huawei

We also interviewed a dedicated UI/UX designer at Huawei, with limited programming experience. His primary job is to create mock-up artifacts that stipulate designs of mobile apps, communicate these to developers, and ensure they are implemented to spec. This interview was translated from Chinese into English. This designer also expressed interest in REDRAW, stating that the visual similarity of the apps was impressive for an automated approach, *"Regarding visual, I feel that it's very similar"*, and that such a solution would be sought after at Huawei, *"If it [a target app] can be automatically implemented after the design, it should be the best design tool [we have]"*. While this designer does not have extensive development experience, he works closely with developers and stated that the quality of the reusability of the code is a key point for adoption, *"In my opinion, for the developers it would be ideal if the output code can be reused"*. This is promising as REDRAW was shown to generate GUI-hierarchies that are comparatively more similar to real apps than other approaches.

5.4.3 Mobile Researcher @Facebook

The last participant was a mobile systems researcher at Facebook. This participant admitted that Facebook would most likely not use REDRAW in its current state, as they are heavily invested in the React Native ecosystem. However, he saw the potential of the approach if it were adopted for this domain, stating *"I can see this as a possible tool to prototype designs"*. He was impressed by the visual similarity of the apps, stating, *"The visual similarity seems impressive"*.

In the end, we can answer **RQ4**:

> **RQ4:** REDRAW **has promise for application into industrial design and development workflows, particularly in an evolutionary context. However, modifications would most likely have to be made to fit specific workflows and prototyping toolchains.**

6 LIMITATIONS & THREATS TO VALIDITY

In this section we describe some limitations and possible routes for future research in automated software prototyping, along with potential threats to validity of our approach and study.

6.1 Limitations and Avenues for Future Work

While REDRAW is a powerful approach for prototyping GUIs of mobile apps, it is tied to certain practical limitations, some of which represent promising avenues for future

work in automated software prototyping. First, REDRAW is currently capable of prototyping a single screen for an application, thus if multiple screens for a single app are desired, they must be prototyped individually and then manually combined into a single application. It would be relatively trivial to modify the approach and allow for multiple screens within a single application with a simple swipe gesture to switch between them for software demo purposes however, we leave this a future work. Additionally, future work might examine a learning-based approach for prototyping and linking together multiple screens, learning common app transitions via dynamic analysis and applying the learned patterns during prototyping. Second, the current implementation of KNN-hierarchy construction is tied to the specific screen size of the devices used during the data-mining and automated dynamic analysis. However, it is possible to utilize display independent pixel (`dp` values) to generalize this algorithm to work independent of screen size, however we leave this as future work. Our current CNN classifier is capable of classifying incoming images into one of 15 of the most popular Android GUI-components. Thus, we do not currently support certain, rarely used component types. Future work could investigate network architectures with more capacity (*e.g.*, deeper architectures) to classify larger numbers of component types, or even investigate non-standard architectures such as Hierarchical CNNs [103]. Currently, REDRAW requires two steps for *detecting* and *classifying* components, however, future approaches could examine the applicability of CNN-based object detection networks [104], [105] that may be capable of performing these two steps in tandem.

6.2 Internal Validity

Threats to *internal validity* correspond to unexpected factors in the experiments that may contribute to observed results. One such threat stems from our semi-structured interview with industrial developers. While evaluating industrial applicability of REDRAW, threats may arise from our manual reverse engineering of Sketch mock-ups. However, we applied a state of art tool for detecting design violations in GUIs [10] in order to ensure their validity, sufficiently mitigating this threat. Another threat arises from using screenshots of existing apps as a proxy for design mockup artifacts in our experimental procedure for **RQ2-RQ3**. As stated earlier, it was not practically feasible to reverse-engineer mock-ups for all 83 applications utilized in our dataset for these experiments. Furthermore, these screenshots represent production-grade app designs that are used daily by millions of users, thus we assert that these screenshots represent a reasonable evaluation set for REDRAW. We also did not observe any confounding results when applying REDRAW to our nine reverse engineered Sketch mock-ups, thus we assert that this threat to validity is reasonably mitigated. Another potential confounding factor is our dataset of labeled components used to train, validate, and test the CNN. To help ensure a correct, coherent dataset, we applied several different data filtering, cleaning, and augmentation techniques, inspired by past work on image classification using CNNs described in detail in Sec. 3.2.4. Furthermore, we utilized the `uiautomator` tool included in the Android

SDK, which is responsible for reporting information about runtime GUI-objects, and is generally accurate as it is tied directly to Android sub-systems responsible for rendering the GUI. To further ensure the validity of our dataset, we randomly sampled a statistically significant portion of our dataset and manually inspected the labeled images *after* our data-cleaning process was applied. We observed no irregularities and thus mitigating a threat related to the quality of the dataset. Furthermore, we will release our full dataset and code for training the CNN upon acceptance of this paper to promote reproducibility and transparency [38].

6.3 Construct Validity

Threats to *construct validity* concern the operationalization of experimental artifacts. One potential threat to construct validity lies in our reimplementation of the REMAUI tool. As stated earlier, the original version of REMAUI's web tool was not working at the time of writing this paper. We reimplemented REMAUI according to the original description in the paper, however we excluded the list generation feature, as we could not reliably re-create this feature based on the provided description. While our version may vary slightly from the original, it still represents an unsupervised CV-based technique against which we can compare REDRAW. Furthermore, we will offer our reimplementation of REMAUI (a Java program with opencv [106] bindings) as an open source project upon acceptance of this paper [38] to facilitate reproducibility and transparency in our experimentation. Another potential threat to construct validity lies in our operationalization of the pix2code project. We closely followed the instructions given in the README of the pix2code project on GitHub to train the machine translation model and generate prototype applications. Unfortunately, the dataset used to train this model differs from the large scale dataset used to train the REDRAW CNN and inform the KNN-hierarchy construction, however, this is due to the fact pix2code requires the source code of training applications and employs a custom domain specific language, leading to incompatibilities to our dataset. We include the pix2code approach as a comparative baseline in this paper as it is one of the few approaches aimed at utilizing ML to perform automated GUI prototyping, and utilizes an architecture based purely upon neural machine translation, differing from our architecture. However, it should be noted that if trained on a proper dataset, with more advanced application assembly techniques, future work on applying machine translation to automated GUI-prototyping may present better results than those reported in this paper for pix2code.

6.4 External Validity

Threats to *external validity* concern the generalization of the results. While we implemented REDRAW for Android and did not measure its generalization to other domains, we assert that implementing our approach for other software domains constitutes mainly engineering effort. This is tied to the fact that other GUI-frameworks are typically comprised sets of varying types of widgets, and GUI-related information can be automatically extracted via dynamic analysis using one of a variety of techniques including accessibility services [77]. We leave the implementation of the approach underlying REDRAW for other platforms as future work.

7 CONCLUSION & FUTURE WORK

In this paper we have presented a data-driven approach for automatically prototyping software GUIs, and an implementation of this approach in a tool called REDRAW for Android. A comprehensive evaluation of REDRAW demonstrates that it is capable of (i) accurately detecting and classifying GUI-components in a mock-up artifact, (ii) generating hierarchies that are similar to those that a developer would create, (iii) generating apps that are visually similar to mock-up artifacts, and (iv) impacting industrial workflows. In the future, we are planning on exploring CNN architectures aimed at object detection and classification to better support the detection task. Additionally, we are planning on working with industrial partners to integrate REDRAW, and our broader prototyping approach, into their workflows.

REFERENCES

[1]   "Apple app store https://www.apple.com/ios/app-store/."
[2]   "Google play store https://play.google.com/store?hl=en."
[3]   "Why your app's ux is more important than you think http://www.codemag.com/Article/1401041."
[4]   "Adobe photoshop http://www.photoshop.com."
[5]   "The sketch design tool https://www.sketchapp.com."
[6]   A. B. Tucker, *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC, 2004.
[7]   B. Myers, "Challenges of hci design and implementation," *Interactions*, vol. 1, no. 1, pp. 73–83, Jan. 1994. [Online]. Available: http://doi.acm.org.proxy.wm.edu/10.1145/174800.174808
[8]   T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 248–259. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.32
[9]   V. Lelli, A. Blouin, and B. Baudry, "Classifying and qualifying gui defects," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.
[10] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of gui design violations in mobile apps," in *Proceedings of the 40th International Conference on Software Engineering Companion*, ser. ICSE '18. Piscataway, NJ, USA: IEEE Press, 2018, p. to appear.
[11] J. A. Landay and B. A. Myers, "Interactive sketching for the early stages of user interface design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '95. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 43–50. [Online]. Available: http://dx.doi.org/10.1145/223904.223910
[12] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, "How designers design and program interactive behaviors," in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, Sept 2008, pp. 177–184.
[13] "Xcode https://developer.apple.com/xcode/."
[14] "Visual-studio https://www.visualstudio.com."
[15] "Android-studio https://developer.android.com/studio/index.html."
[16] C. Zeidler, C. Lutteroth, W. Stuerzlinger, and G. Weber, *Evaluating Direct Manipulation Operations for Constraint-Based Layout*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 513–529. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40480-1_35
[17] "Mockup.io https://mockup.io/about/."
[18] "Proto.io https://proto.io."
[19] "Fuild-ui https://www.fluidui.com."
[20] "Marvelapp https://marvelapp.com/prototyping/."
[21] "Pixate http://www.pixate.com."
[22] "Xiffe http://xiffe.com."
[23] "Mockingbot https://mockingbot.com."
[24] "Flinto https://www.flinto.com."
[25] "Justinmind https://www.justinmind.com."
[26] "Protoapp https://prottapp.com/features/."

[27] "Irise https://www.irise.com/mobile-prototyping/."

[28] "Appypie http://www.appypie.com/app-prototype-builder."

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[30] M. D. Zeiler and R. Fergus, *Visualizing and Understanding Convolutional Networks*. Cham: Springer International Publishing, 2014, pp. 818–833. [Online]. Available: https://doi.org/10.1007/978-3-319-10590-1_53

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[32] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015. [Online]. Available: http://arxiv.org/abs/1409.4842

[33] K. He, X. Zhang, S. Ren, and J. Sun, in *2016 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR'16.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1007/s11263-015-0816-y

[35] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST'16)*. IEEE, 2016, pp. 33–44.

[36] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *12th Working Conference on Mining Software Repositories (MSR'15)*, 2015, p. to appear.

[37] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," *CoRR*, vol. abs/1705.07962, 2017. [Online]. Available: http://arxiv.org/abs/1705.07962

[38] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Redraw online appendix https://www.android-dev-tools.com/redraw."

[39] "Android ui-development https://developer.android.com/guide/topics/ui/overview.html."

[40] A. Coyette, S. Kieffer, and J. Vanderdonckt, "Multi-fidelity prototyping of user interfaces," in *Proceedings of the 11th IFIP TC 13 International Conference on Human-computer Interaction*, ser. INTERACT'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 150–164. [Online]. Available: http://dl.acm.org/citation.cfm?id=1776994.1777015

[41] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge, "Javasketchit: Issues in sketching the look of user interfaces," in *AAAI Spring Symposium on Sketch Understanding*, ser. SSS'02, 2002, pp. 9–14.

[42] J. A. Landay and B. A. Myers, "Sketching interfaces: toward more human interface design," *Computer*, vol. 34, no. 3, pp. 56–64, Mar 2001.

[43] S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz, "Revisiting visual interface programming: Creating gui tools for designers and programmers," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '04. New York, NY, USA: ACM, 2004, pp. 267–276. [Online]. Available: http://doi.acm.org/10.1145/1029632.1029678

[44] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt, "Mobidev: A tool for creating apps on mobile phones," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI'11)*, ser. MobileHCI '11. New York, NY, USA: ACM, 2011, pp. 109–112. [Online]. Available: http://doi.acm.org/10.1145/2037373.2037392

[45] X. Meng, S. Zhao, Y. Huang, Z. Zhang, J. Eagan, and R. Subramanian, "Wade: Simplified gui add-on development for third-party software," in *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2221–2230. [Online]. Available: http://doi.acm.org.proxy.wm.edu/10.1145/2556288.2557349

[46] W. S. Lasecki, J. Kim, N. Rafter, O. Sen, J. P. Bigham, and M. S. Bernstein, "Apparition: Crowdsourced user interfaces that come to life as you sketch them," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 1925–1934. [Online]. Available: http://doi.acm.org/10.1145/2702123.2702565

[47] T.-H. Chang, T. Yeh, and R. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. New York, NY, USA: ACM, 2011, pp. 245–256. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047228

[48] M. Dixon, D. Leventhal, and J. Fogarty, "Content and hierarchy in pixel-based methods for reverse engineering interface structure," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 969–978. [Online]. Available: http://doi.acm.org/10.1145/1978942.1979086

[49] M. Dixon and J. Fogarty, "Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1525–1534. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753554

[50] A. Hinze, J. Bowen, Y. Wang, and R. Malik, "Model-driven gui & interaction design using emulation," in *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '10. New York, NY, USA: ACM, 2010, pp. 273–278. [Online]. Available: http://doi.acm.org/10.1145/1822018.1822061

[51] E. Shah and E. Tilevich, "Reverse-engineering user interfaces to facilitateporting to and across mobile devices and platforms," in *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011, pp. 255–260. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095093

[52] H. Samir and A. Kamel, "Automated reverse engineering of java graphical user interfaces for web migration," in *2007 ITI 5th International Conference on Information and Communications Technology*, ser. ICICT'07, Dec 2007, pp. 157–162.

[53] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[54] "The marketch plugin for sketch https://github.com/tudou527/marketch."

[55] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov 1986.

[56] "Sketch extensions https://www.sketchapp.com/extensions/."

[57] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'13. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[58] "Android ui/application exerciser monkey http://developer.android.com/tools/help/monkey.html."

[59] "Intent fuzzer https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx."

[60] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics*, ser. WODA+PERTEA'14. New York, NY, USA: ACM, 2014, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/2632168.2632169

[61] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[62] 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*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[63] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE'12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[64] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'12. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[65] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 15–18. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.16

[66] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[67] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[68] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[69] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 183–192. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.31

[70] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA'16. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[71] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'14. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[72] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: A web application testing tool for cross-browser inconsistency detection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 417–420. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628057

[73] J. Thomé, A. Gorla, and A. Zeller, "Search-based security testing of web applications," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, ser. SBST 2014. New York, NY, USA: ACM, 2014, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/2593833.2593835

[74] S. Roy Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609723

[75] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.97

[76] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10515-013-0128-9

[77] M. Grechanik, Q. Xie, and C. Fu, "Creating gui testing tools using accessibility technologies," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW'09, April 2009, pp. 243–250.

[78] "Android uiautomator http://developer.android.com/tools/help/uiautomator/index.html."

[79] "Google-api https://github.com/NeroBurner/googleplay-api."

[80] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *in Proceedings of 10th Joint Meeting of the European Software Engineering Conference and the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Bergamo, Italy, Ausgust-September 2015 2015, p. to appear.

[81] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, 2015.

[82] 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, 2014.

[83] J. Gui, S. Mcilroy, M. Nagappan, and W. G. Halfond, "Truth in advertising: The hidden cost of mobile ads for software developers," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE'15. Florence, Italy: IEEE Press, 2015, pp. 100–110.

[84] rovo89, "Xposed module repository http://repo.xposed.info/."

[85] U. Karn, "An intuitive explanation of convolutional neural nets https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/."

[86] "Convolution operator http://mathworld.wolfram.com/Convolution.html."

[87] "Apache cordova https://cordova.apache.org."

[88] "apktool https://code.google.com/p/android-apktool/."

[89] "Unity game engine https://unity3d.com."

[90] "Matlab neural network toolboxhttps://www.mathworks.com/products/neural-network.html."

[91] "Tesseract ocr library https://www.mathworks.com/products/parallel-computing.html."

[92] "Tesseract ocr library https://www.mathworks.com/products/computer-vision.html."

[93] "Tesseract ocr library https://github.com/tesseract-ocr/tesseract/wiki."

[94] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'17. New York, NY, USA: ACM, 2017, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106256

[95] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on statistical learning in computer vision*, ser. ECCV'04, vol. 1, no. 1-22. Prague, 2004, pp. 1–2.

[96] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cviu.2007.09.014

[97] "Pix2code github repository https://github.com/tonybeltramelli/pix2code."

[98] "Remaui web version http://pixeltoapp.com."

[99] "Wagner-fischer algorithm https://en.wikipedia.org/wiki/WagnerFischer_algorithm."

[100] "Photohawk library http://datascience.github.io/photohawk/."

[101] W. Conover, *Practical Nonparametric Statistics*. Wiley, 1998.

[102] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2005.

[103] Z. Wang, X. Wang, and G. Wang, "Learning fine-grained features via a CNN tree for large-scale classification," *CoRR*, vol. abs/1511.04534, 2015. [Online]. Available: http://arxiv.org/abs/1511.04534

[104] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 91–99. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969239.2969250

[105] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 580–587. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2014.81

[106] "Opencv https://opencv.org."

Kevin Moran is currently a Ph.D. candidate in the Computer Science Department at the College of William and Mary. He is a member of the SEMERU research group and advised by Dr. Denys Poshyvanyk. His main research interest involves facilitating the processes of Software Engineering, Maintenance, and Evolution with a focus on mobile platforms. He graduated with a M.S. degree from William and Mary in August of 2015 and his thesis focused on improving bug reporting for mobile apps through novel applications of program analysis techniques. He has published in several top peer-reviewed software engineering venues including: ICSE, ESEC/FSE, ICST, ICSME, and MSR. He was recognized as the second-overall graduate winner in the ACM Student Research competition at ESEC/FSE15. Moran is a student member of IEEE and ACM and has served as an external reviewer for ICSE, ICSME, FSE, APSEC, and SCAM. More information available at http://www.kpmoran.com.

Carlos Bernal-Cárdenas received the BS degree in systems engineering from the Universidad Nacional de Colombia in 2012 and his M.E. in Systems and Computing Engineering in 2015. He is currently Ph.D. candidate in Computer Science at the College of William and Mary as a member of the SEMERU research group advised by Dr Denys Poshyvanyk. His research interests include software engineering, software evolution and maintenance, information retrieval, software reuse, mining software repositories, mobile applications development, and user experience. He has published in several top peer-reviewed software engineering venues including: ICSE, ESEC/FSE, ICST, and MSR. He has also received the ACM SigSoft Distinguished paper award at ESEC/FSE'15. Bernal-Cárdenas is a student member of IEEE and ACM and has served as an external reviewer for ICSE, ICSME, FSE, APSEC, and SCAM. More information is available at http://www.cs.wm.edu/~cebernal/.
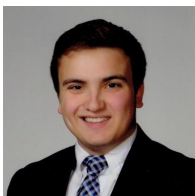
Michael Curcio is an undergraduate student in the Computer Science Department at the College of William and Mary. He is currently a member of the SEMERU research group and is pursuing an undergraduate honors thesis on the topic of automating software design workflows. His research interests lie in applications of deep learning to software engineering and design tasks. Curcio is an IEEE student member.

Richard Bonett is a MS/PhD student at The College of William and Mary and a member of the SEMERU research group. He graduated from The College of William and Mary with a B.S. in Computer Science in Spring 2017. His primary research interests lie in Software Engineering, particularly in the development and evolution of mobile applications. Bonett has recently published at MobileSoft'17. More information is available at http://www.cs.wm.edu/~rfbonett/.
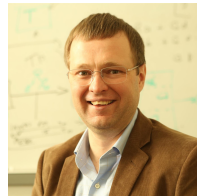
Denys Poshyvanyk is an Associate Professor in the Computer Science Department at W&M where he leads SEMERU research group. He received his Ph.D. from Wayne State University, where he was advised by Dr. Andrian Marcus. His current research is in the area of software engineering, evolution and maintenance, program comprehension, reverse engineering, software privacy, repository mining, traceability, performance testing, mobile app (Android) development and testing, energy consumption, and reuse. He has received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16, and ASE'17. He is also a recipient of the NSF CAREER award (2013) and the Plumeri Award for Faculty Excellence (2016). More information available at: http://www.cs.wm.edu/~denys.