GUIFetch: Supporting App Design and Development through GUI Search

Farnaz Behrang
Georgia Tech
Atlanta, GA
behrang@gatech.edu

Steven P. Reiss
Brown University
Providence, RI
spr@cs.brown.edu

Alessandro Orso
Georgia Tech
Atlanta, GA
orso@cc.gatech.edu

ABSTRACT

A typical way to design and develop a mobile app is to sketch the graphical user interfaces (GUIs) for the different screens in the app and then create actual GUIs from these sketches. Doing so involves identifying which layouts to use, which widgets to add, and how to configure and connect the different pieces of the GUI. To help with this difficult and time-consuming task, we propose GUIFetch, a technique that takes as input the sketch for an app and leverages the growing number of open source apps in public repositories to identify apps with GUIs and transitions that are similar to those in the provided sketch. GUIFetch first searches public repositories to find potential apps using keyword matching. It then builds models of the identified apps' screens and screen transitions using a combination of static and dynamic analyses and computes a similarity metric between the models and the provided sketch. Finally, GUIFetch ranks the identified apps (or parts thereof) based on their computed similarity value and produces a visual ranking of the results together with the code of the corresponding apps. We implemented GUIFetch for Android apps and evaluated it through user studies involving different types of apps.

CCS CONCEPTS

• **Software and its engineering** → Software design engineering;

KEYWORDS

User interface design, user interface programming, code search

ACM Reference Format:
Farnaz Behrang, Steven P. Reiss, and Alessandro Orso. 2018. GUIFetch: Supporting App Design and Development through GUI Search. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3197231.3197244

1 INTRODUCTION

There is an ever growing amount of code available and easily accessible online in public repositories, such as GitHub (www.github.com) and Bitbucket (https://bitbucket.org/). It is therefore not surprising that, in the last few years, we have witnessed an increasing interest in code-search techniques (e.g., [1, 26, 44]). More recently, researchers have moved beyond searching for small snippets of code and have developed sophisticated techniques that allow developers to search for larger fragments of code (e.g., [4, 5, 37]) and even for graphical user interface (GUI) code [38]. In this paper, we push this concept even further and present a code-search technique that can support designers and developers of mobile apps.

A typical way employed by many industrial companies to design and develop a mobile app is to go from the sketch of an app to the actual app [24, 32]. The process starts with first sketching the GUIs for the different screens of the app and then creating actual GUIs from the sketches. Finally, developers connect the GUIs to the code that implements the app's functionality. In this paper, we focus on the part of this process that goes from sketches to GUIs, which involves identifying which layouts to use, which widgets to add, and how to configure and connect the different pieces of the GUIs. To help in this difficult and time consuming task, we propose GUIFetch, a technique that takes advantage of the growing number of open source apps in public repositories (e.g., there are more than 300,000 [27] Android apps on GitHub) to provide users with GUIs and transitions that are similar to those in their provided sketch.

None of the existing code-search techniques would fully work for mobile apps. These techniques require code snippets or a single GUI as the starting point, whereas the sketch of an app might consist of different screens and transitions. Moreover, creating the model of app GUIs is particularly challenging, as there is not just a single way to build them. Specifically, users can take advantage of XML layout resources, create GUIs programmatically by using GUI library functions, or use combinations of both approaches. The bottom line is that different code-search techniques are required to support code search in the context of mobile apps.

Given a set of sketches for the various screens in an app, transitions between the screens corresponding to the sketches, and a set of keywords, GUIFetch first searches public repositories to find potential apps using keyword matching. It then builds models of the identified apps' screens and screen transitions, using a combination of static and dynamic analyses, and computes a similarity metric between the models and the user provided sketches. Finally, it ranks the identified apps (or parts thereof) based on their computed similarity value and presents the ranked results visually to the users, together with the code for the corresponding apps.

GUIFetch can be used in different scenarios in which traditional GUI builder would fall short. App designers and developers can use the approach to explore the state of the art of an application domain; seeing a GUI that is similar in spirit to (but slightly different from)

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197244

Figure 1: Sketch of an expense tracking app with two screens and three transitions (Screen A $\xrightarrow{AddNewExpense}$ Screen B, Screen B $\xrightarrow[Cancel]{Save}$ Screen A).

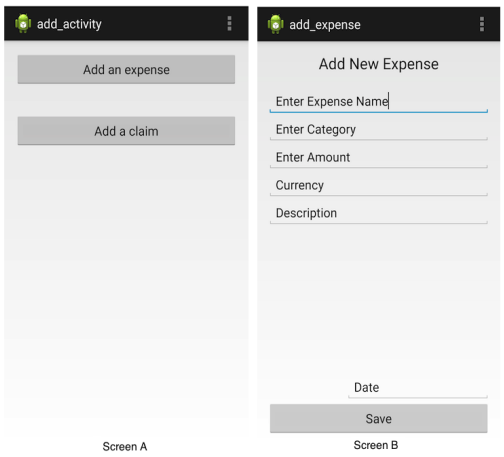

Figure 2: App returned by GUIFetch as the top match for the sketch in Figure 1 (app transitions: Screen A $\xrightarrow{Addanexpense}$ Screen B, Screen B $\xrightarrow{Save}$ Screen A).

the one sketched may provide new ideas to developers on how to refine their sketches. In addition, generating GUIs often happens as part of an iterative development approach, where GUIs are constantly updated or changed (e.g., within agile development [13]). In this context, GUIFetch can be used to support early prototyping, in that developers could find a GUI that is close enough to the one they are planning to develop, possibly modify it, and use it as an initial throw-away prototype. Finally, in the case in which the quality of the code returned by GUIFetch meets the expectation, developers can use that as a starting point for building their actual apps.

We implemented GUIFetch for Android apps and empirically evaluated the approach. First, we used our tool to find GUI code for six different types of mobile apps and successfully checked that GUIFetch reported the app most similar to the provided sketch as its top match. Second, we conducted a user study involving 16 participants. In the study, we (1) asked each participant to sketch an app of a given (randomly selected) type, (2) ran GUIFetch using the sketch as input, (3) presented GUIFetch's recommendations to the participant, and (4) asked the participants to use GUIFetch's recommendations and assess their usefulness. The results of the user study are promising. Based on the participants' feedback, 81% of GUIFetch's recommendations were relevant to the participants' sketches, and GUIFetch's ranking correlated with the ranking provided by the participants in many cases. Moreover, participants were able to use GUIFetch's recommendations to make changes to their sketches by adding, updating, or deleting GUI elements. Finally, the responses we collected through a questionnaire gave us insights about potential additional applications of GUIFetch, as well as weaknesses that we can address in future work.

Our paper makes the following contributions:

- GUIFetch, a new code-search technique that takes a user-provided sketch of an app and fetches relevant code from public repositories.
- A publicly available implementation of GUIFetch [14].
- An empirical evaluation that provides initial evidence of the effectiveness and feasibility of our approach.

The rest of this paper is organized as follows. Section 2 presents a motivating example. Section 3 discusses our technique. We present the results of our empirical evaluation in Section 4. Finally, Sections 5 and 6 discuss related work and provide some concluding remarks on the work we presented and on possible future research directions.

2 MOTIVATING EXAMPLE

As an example, assume a developer or a GUI designer uses a sketching tool to draw the sketch shown in figure 1, which represents an expense tracking app. The sketch consists of two screens with three possible transitions (shown with arrows in the sketch). The first screen shows the total expenses for the current month and has two buttons: *View Expense* and *Add New Expense*. Clicking the *Add New Expense* button opens the second screen, which allows the user to enter the expense details, such as date, description, cost, and type. Clicking on either *Cancel* or *Save* brings the user back to the first screen.

Given a set of keywords and the sketch, GUIFetch would search in one or more open source code repositories and return to the user a set of relevant apps. To illustrate, Figure 2 shows the screens of an app that was returned by GUIFetch as the top match for this sketch in our evaluation. As the figure shows, the app also contains two screens. Moreover, similar to the sketch, clicking on button *Add an expense* takes the users to the second screen and allows them to enter the details of an expense. Clicking on the *Save* button, conversely, causes the app to go back to the first screen.

GUIFetch would return the screenshots of the app (shown in figure 2), along with the app source code. Providing the screenshots, in addition to the source code, makes it easier for the users to perform a first screening of the results, without any need for compiling or running the code.

Figures 1 and 2 clearly show that, although the sketch and the app are not identical, they are conceptually fairly similar. In particular, there is a mapping between the elements of the sketch and the widgets of the app. Also, two out of three transitions in the user provided sketch exist in the app.

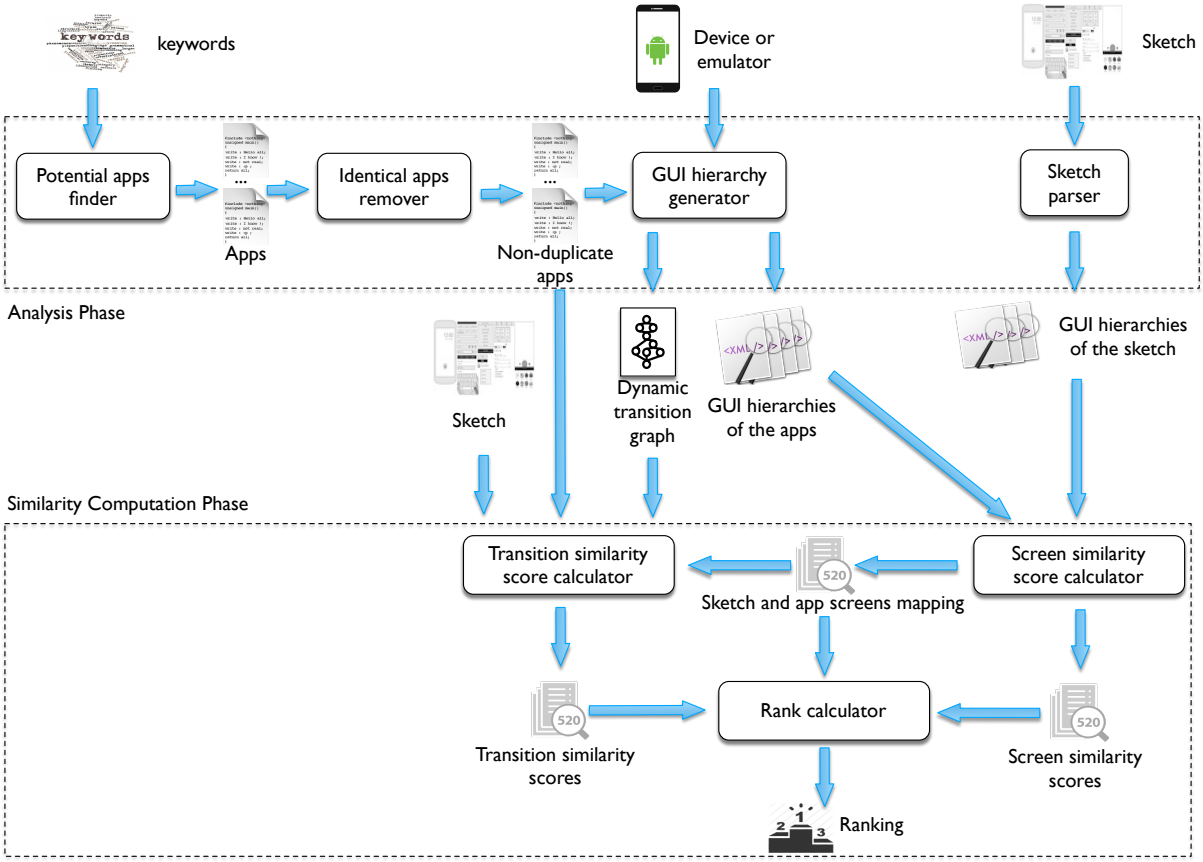

Figure 3: Overview of GUIFetch.

It is worth noting that, in addition to the similar elements that exist in both the sketch and the app, there are two more elements in the app—related to expense name and currency—that do not exist in the sketch. If the designer or the developer found these elements relevant for what they had in mind, they could add them to their design by just looking at the screenshots. They would also have the opportunity to take a look at the source code to see if they are interested in using it as an initial prototype or as a starting point for building the app.

3 TECHNIQUE

Figure 3 shows an overview of our technique, GUIFetch. As the figure shows, the technique is divided into two phases: *Analysis* and *Similarity Computation*.

Given a set of keywords and a sketch of an app, the *Analysis Phase* is responsible for finding the potential apps from open source code repositories, removing duplicates, and analyzing the source code and the sketch. The outputs of this phase are GUI hierarchies for every possible screen of the apps and the sketch, together with transition graphs for the apps. The *Similarity Computation Phase* uses these generated GUI hierarchies and transition graph to (1) compute a similarity score between the sketch and the apps and (2) provide a ranking based on the computed similarity scores. In the rest of this section, we explain each phase in more detail.

3.1 Analysis Phase

3.1.1 Potential Apps Finder. The first stage of the analysis phase involves finding relevant apps in an open-source repository given a set of keywords related to the app of interest.

The search process starts with the keywords and performs two separate searches. The first search looks for Java source files containing the keywords as well as the term "Android". Source files are considered here because they are more likely to contain comments that describe the application and that might be a good match for the user-provided keywords. The second search looks for Android manifest files that contain the keywords. More precisely, this search focuses on XML files that contain the original keywords as well as the terms "Android", "manifest", "application", and "activity". For each of these searches, GUIFetch looks at the first 100 matching files returned by GitHub. (We currently use GitHub as the search engine since the underlying repository includes not only the source files but also all the related Android resource files which will be needed to run the program.) GUIFetch treats each returned file as an indicator of what project should be considered. It then tries to compile the corresponding projects and considers as potential apps those that compile successfully.

3.1.2 Identical Apps Remover. Retrieved code from open source repositories is likely to include identical or highly similar apps.

GUIFetch removes duplicates to reduce the running time of the technique and make the final ranking more useful to users. To do so, GUIFetch uses an existing plagiarism detection technique [36]. Although this technique does not consider files other than source code (e.g., XML files), it is highly unlikely for two apps to have extremely similar GUIs and different source code. GUIFetch removes all the apps with a similarity value higher than 70%, a threshold computed based on preliminary experiments.

3.1.3 Sketch Parser. To make the user sketch comparable to real apps, GUIFetch needs to generate GUI hierarchies for it, along with transitions between the GUIs. The first step in generating comparable GUI hierarchies is to find a mapping between the types of Android widgets and the type of elements in the sketch. For screenshots or conceptual drawings, for instance, this is achievable using OCR and computer vision techniques [31], whereas various heuristics can be used for SVG-based sketches [38]. For some prototyping tools, in particular, it might be possible to directly parse their output, as we do in our current implementation (see Section 4.1.

After finding the mapping between widget types, GUIFetch extracts all the elements of the sketch along with their attributes. These attributes include type, height, width, dimensions, and any text associated with the widgets. (For dimensions and coordinates, values relative to the sketch size are used.) Then, GUIFetch extracts from the sketch the transitions between screens provided by the user and builds a graph where nodes are screens and edges are transitions labeled with the widgets that cause the transition.

3.1.4 GUI Hierarchy Generator. After removing identical apps and parsing the sketch, GUIFetch generates the GUI hierarchies for every possible screen of the app. In Android, GUIs can be built both statically and dynamically. Specifically, GUIs (or parts thereof) can be built statically through XML layout files or dynamically through calls to library functions. Therefore, to identify all screens and their corresponding widgets, GUIFetch needs to perform both static and dynamic analyses; (1) a purely static approach would miss those parts of the GUI that are built dynamically, whereas (2) a purely dynamic approach may not be able to reach, and thus model, all screens and widgets therein.

The overview of the *GUI Hierarchy Generator* is shown in Figure 4. GUIFetch first gets the source code of the app and finds a mapping between screen layout files and source files. In this way, it identifies all the possible screens of the app and detects which layout and source files correspond to which screen. Note that, although this analysis identifies all screens, the widgets in the screens can be modified at runtime. For this reason, the GUI hierarchy that is obtained dynamically, when the dynamic analysis is successful, is generally more precise than the one computed statically.

After identifying screens and mappings to layout and source files, GUIFetch performs a dynamic analysis based on crawling—it launches the app and generates events, such as clicks, long clicks, and so on, through the user interface trying to reach all the screens in the app. Every time a new screen is observed, GUIFetch records it. Since transitions between screens also matter, GUIFetch keeps track of such transitions during the analysis. Finally, GUIFetch models the execution of the app as a transition graph where nodes represent screens and edges represent transitions between screens.

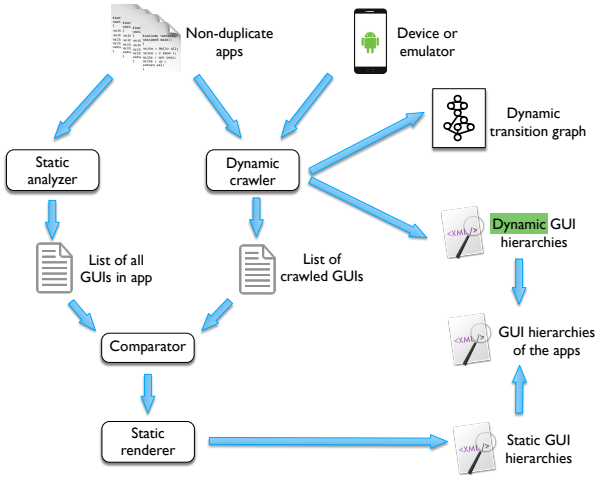

Figure 4: Overview of the GUI hierarchy generator.

The outputs of the dynamic crawler are the GUI hierarchies of each screen identified and a graph of transitions between them. GUIFetch then compares the set of screens identified dynamically with the set of all screens determined initially. For every screen that was not reached dynamically, it uses a static renderer that generates a screen based on the corresponding layout file and gets the GUI hierarchy for that screen. The final output of this step is the combination of the GUI hierarchies collected statically and dynamically.

3.2 Similarity Computation Phase

In this phase, GUIFetch takes as input the source code of non-duplicate apps, the GUI hierarchies of the screens of these apps and of the sketch, and the dynamic transition graph it computed in the analysis phase. Given this input, GUIFetch computes an *overall similarity* score between each app considered and the sketch, as follows. First, GUIFetch computes the similarity between each screen in the app and each screen in the sketch (*screen similarity*, see Section 3.2.1). Second, GUIFetch defines a mapping between screens in the app and screens in the sketch based on their corresponding screen similarity score. It then uses this mapping to compute a measure of how well the transitions in the app and in the sketch match (*transition similarity*, see Section 3.2.2). Finally, GUIFetch adds the various screen similarity scores (for all the screens that match) and the transition similarity score to compute the overall similarity score. We now describe in more detail these steps.

3.2.1 Screen Similarity Score Calculator. The inputs to this step are two GUI hierarchies: one for a screen of the sketch and the other for a screen of an app. GUIFetch compares these GUI hierarchies by widget and by considering four criteria for every widget:[1] type, associated text (if any), size (width and height), and position. We choose these four criteria as they were shown to be effective in earlier work [38]. In the rest of this section, we discuss how each of these criteria affects the similarity score of two widgets. Note that the percentages that we assign to the different criteria are based

[1]For ease of explanation, hereafter we refer to both the actual widgets in the app and the elements in the sketch as *widgets* and use the terms *app widget* and *sketch widget*, respectively, to refer to them.

on our preliminary empirical investigation, in which we trained GUIFetch on about 100 apps and tested it on 40 different apps to identify effective thresholds.

GUIFetch performs *type matching* between sketch widgets and app widgets using the mapping between sketch elements and Android widget types it computed while parsing the sketch (see Section 3.1.3). For each sketch widget, GUIFetch considers all potentially matching app widgets and computes a similarity score as follows. If both widgets contain text, GUIFetch performs *text matching*; Otherwise, it moves to the size and position matching. To perform text matching, GUIFetch first applies some heuristics that aim to improve precision. Specifically, GUIFetch uses a POS (Part-Of-Speech) tagger to assign POS tags to each word in the text, such as noun, verb, adjective, and so on. In this way, GUIFetch can ignore irrelevant elements, such as pronouns, conjunctions, and prepositions, when comparing text. For instance, a header in a sketch screen with title "AddressBook" would match the header in an app screen with title "My AddressBook", as the pronoun "My" would be ignored. GUIFetch also converts all the characters in the text to lower case. After this preprocessing, GUIFetch computes the Levenshtein distance [3] between the two text elements, assigns a score to this match accordingly, and checks whether the score is above a given threshold. If not, GUIFetch sets the similarity score to zero and stops considering the pair as a possible match. Otherwise, it normalizes the score to 60 (i.e.,text matching constitutes 60% of the overall similarity score) and continues. (We chose this value based on preliminary experimentation, as we discussed earlier in this section.)

In the *size and position matching*, GUIFetch first normalizes the width, height, x coordinate, and y coordinate of app and sketch widget relatively to the screen size. Then, it computes the differences for each of this four normalized values between the two widgets, computes four separate scores proportional to such differences, and normalizes each score to 10 (i.e.,each score constitutes 10% of the overall similarity score). Also in this case, if the resulting similarity is below a given threshold, GUIFetch sets the similarity score to zero and stops considering the pair as a possible match. Otherwise, it either adds the computed similarity to the text matching similarity or further normalize the similarity score to 100, if no text matching was performed.

After all the potentially matching app widgets for a sketch widget have been evaluated, GUIFetch selects as a match the app widget that has the highest similarity score. Finally, when all the widgets on the input sketch screen have been processed, GUIFetch computes the overall similarity score between this screen and the input app screen as the ratio consisting of the sum of the similarity values for all the sketch widgets over the maximum possible value for the sum (i.e.,100 times the number of sketch widgets).

3.2.2 Transition Similarity Score Calculator. As we mentioned earlier, besides measuring the similarity of screens, GUIFetch also computes a similarity score for the transitions between screens. Figure 5 shows an overview of how GUIFetch computes such *transition similarity score*.

The analysis phase of GUIFetch (see Section 3.1) produces a graph of screens and transitions derived from the dynamic analysis. To compute transition similarity scores, GUIFetch complements

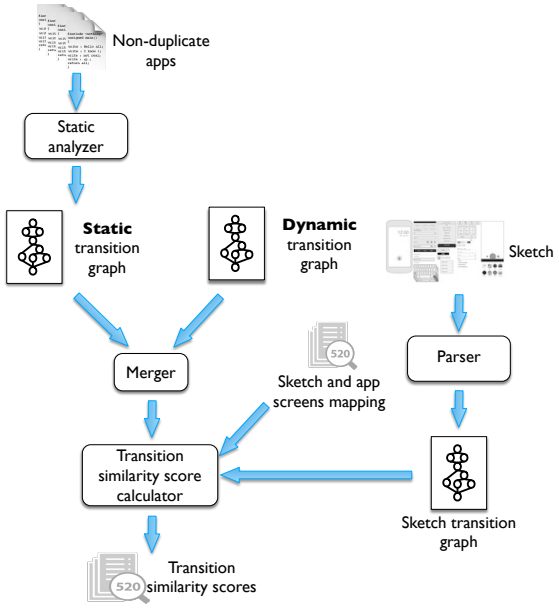

Figure 5: Overview of the transition similarity score calculator.

this dynamically computed information with information computed through static analysis. Specifically, our technique uses static analysis to compute a static transition graph and then merges these the dynamic and static graphs into a single graph. To merge the graphs, GUIFetch adds all the nodes and transitions of the dynamic transition graph to the static transition graph if the nodes or transitions do not already exist. To compare nodes and check for their existence, GUIFetch compares the GUIHierarchy of the corresponding screens. Conversely, GUIFetch compares transitions by comparing their labels, which correspond to the widgets that caused the transitions. After merging dynamic and static transition graphs, GUIFetch compares the merged graph with the corresponding transition graph derived from the sketch. The goal of this comparison is to check whether the transitions defined by the user in the sketch exist in the app. Therefore, for every transition between two nodes of the sketch transition graph, GUIFetch checks if such a transition exists in the app transition graph, using the mapping between screens in the app and screens in the sketch determined in section 3.2.1. If the transition exists, GUIFetch assigns the (previously computed) similarity score for the two widgets that cause the transition in the two graphs as the similarity score for the transition. Conversely, if the transition does not exist, its similarity score is considered to be zero.

3.2.3 Rank Calculator. After computing screen and transition similarity scores for all the screens and transitions in the sketch, GUIFetch (1) computes the overall similarity score by adding these scores and (2) ranks the apps based on this value. It is worth noting that the ranking at the app level might not be effective in all cases. Suppose, for instance, an app that has one screen with a very low (or a very high) screen similarity score compared to other screens' similarity scores. In this case, the similarity score for this screen could dramatically change the ranking of the app. To make users aware of this possible effect, besides showing the ranking at the

App category	Apps#	Scr#/Trans# (App 1)	Scr#/Trans# (App 2)
address book	29	1/0	2/2
expense tracking	28	3/6	4/4
note taking	25	2/4	3/6
fitness	22	5/11	6/6
mobile banking	18	4/7	6/8
online shopping	23	5/5	7/11
Total	145	-	-

Table 1: Apps statistics.

app level, GUIFetch also provides the ranking of each app's screen per sketch screen. In this way, users can see both rankings, side by side and decide which one works better for them.

Since the keyword search can return apps that are completely irrelevant or simply too different from the sketch, it is important to identify and discard these apps, which would only introduce noise and overwhelm the user. To do so, GUIFetch keeps track of the difference between every two consecutive overall similarity scores in the ranking and stops the ranking at the point where the largest gap occurs.

4 EVALUATION

To assess effectiveness and usefulness of our approach, we developed a tool that implements GUIFetch and used the tool to investigate four research questions:

(1) **RQ1:** Are GUIFetch recommendations accurate?
(2) **RQ2:** How relevant are GUIFetch recommendations for users?
(3) **RQ3:** How well does GUIFetch ranking match user ranking?
(4) **RQ4:** Do GUIFetch recommendations provide users with insights on how they could improve their initial sketches?

We investigated RQ1 through an analytical study (Section 4.2) and RQ2, RQ3, and RQ4 through a user study. In the rest of this section, we describe in detail our implementation, our empirical study, and our user study.

4.1 Implementation

Our implementation of GUIFetch supports Android applications. We chose Android because it is one of the major platforms in the mobile application market and because there is a large number of open source apps available for this platform. Our approach, however, is general and could be implemented for other platforms.

To implement the apps search, we leveraged the open source framework S^6 [37], which we modified to better handle searches involving Android apps. In particular, we added to S^6 the ability to search for manifest files, in addition to source code, and interpret the manifest contents in terms of source files referenced therein.

To support app sketches with multiple screens and screen transitions, we implemented in our tool support for *Pencil* [33], an open-source GUI prototyping tool. *Pencil* can be run as a standalone application or within the Firefox browser, as a plugin. It provides 73 different Android specific widgets and allows users to define transitions from any widget to any screen. Although the sketch can be exported in several formats, we chose to parse the *Pencil* document file, which is in XML format, Our tool could easily be extended to support other formats or other kinds of sketches.

We implemented our dynamic analysis on top of Espresso [12], a testing framework that provides APIs for writing GUI tests that simulate user interactions with an app. Our dynamic crawler is inspired by PUMA [16], which is a programmable GUI-automation framework for dynamic analysis. For our static analysis, we leveraged gator [47], a static analysis tool that creates a model of the GUI-related behavior of an Android app, as well as a model of the app's control flow. We used gator to find the mappings between layouts and source files and to generate static transition graphs for an app. We also leveraged the rendering engine of Android Studio [2] to render an app's layout files and retrieve the corresponding GUI hierarchies.

4.2 Analytical Study (RQ1)

To answer RQ1 we studied whether, given a sketch (S) and a set of apps one of which (A) matches the sketch, GUIFetch would successfully recommend A as the best match for S. Specifically, we selected six different types of apps, namely, address book, expense tracking, note taking, fitness, mobile banking, and online shopping apps. We chose these categories because they represent a diverse range of apps and because are well represented in GitHub (i.e., they are good candidates for evaluating GUIFetch's ability to find and rank apps). Table 1 provides, for each of the app categories that we studied (Column "App category), the total number of apps that GUIFetch found for that category through keyword search and after eliminating duplicates (Column "App#"). As keywords, we simply used the name of the categories along with the word "Android".

To create the sketches needed for the study, we randomly selected two apps for each app category considered among the ones we found, ran them manually to identify their screens and transitions, and created their sketches accordingly using the *Pencil* tool. The two columns labeled "Scr#/Trans#" in Table 1 report the number of screens and transitions for the two randomly selected apps.

For each sketch S we created for each category C, we provided S to GUIFetch and ran it against all the apps in C, including the app randomly selected to generate the sketch. For all twelve sketches, GUIFetch ranked the correct app as the top match for the sketch. To further evaluate GUIFetch's performance, we also confirmed that, for every screen of the sketch, the corresponding screen in the correct app was always the top match.

4.3 User Study

This section describes the user study we conducted to answer the rest of our research questions.

4.3.1 Participants. We recruited 16 participants (6 male, 10 female, aged 19 to 30). The participants were recruited by advertising the study in undergraduate- and graduate-level HCI related courses. We required participants to have either Android app design or Android app development background. 60% of the participants had the former, whereas 40% had both. All participants were given a $25 gift card for their participation.

4.3.2 Protocol. At the beginning of the study, each participant was randomly assigned a category of mobile apps that was either "address book" or "mobile banking". We chose these two categories,

User ID	*Category*	*#Sketch Screens*	*#Recomm. Screens*
1	address book	2	1
2	address book	3	2
3	address book	3	2
4	address book	1	1
5	address book	1	0
6	address book	4	2
7	address book	2	2
8	address book	2	1
Total	-	**18**	**11**
9	mobile banking	7	4
10	mobile banking	4	3
11	mobile banking	5	3
12	mobile banking	3	1
13	mobile banking	4	3
14	mobile banking	3	3
15	mobile banking	5	3
16	mobile banking	5	4
Total	-	**36**	**24**
Total	-	**54**	**35**

Table 2: Number of screens in the user sketches and in GUIFetch recommendations.

also randomly, among the categories we considered in our analytical study (see Section 4.2). We limited the number of categories so as to have an adequate sample size in each category and be able to run statistical tests [9]. We then introduced the participants to the *Pencil* sketching tool and gave them some time to become familiar with the tool and possibly ask us questions about it. After making sure that the participants were comfortable with the tool, we asked them to start drawing the sketch of an Android app related to their randomly assigned category. Participants were allowed to use any resources when drawing the sketch (e.g., online search or apps on their phone). We also provided them with a pencil and a sheet of paper, in case they wanted to draw the sketch on paper first or needed to take notes.

Once participants drew the sketch, they were asked to run the GUIFetch tool with their sketch as input. To reduce the waiting time associated with a real-time GitHub search, in the study we connected GUIFetch with a local database that contained the apps GUIFetch found, for the categories considered, in the context of our analytical study (see Section 4.2). As shown in Table 1, this set consisted of 29 address book and 18 mobile banking apps.

After running GUIFetch and receiving the recommendations it produced, participants were asked to separate the recommendations they found to be relevant for their sketches from the ones they considered irrelevant. Participants were also asked to manually rank the recommendations they received, so that we could compare their rankings with the rankings generated by GUIFetch.

To support these tasks, we developed a web application designed to guide the participants through the different steps of the study. The application provided instructions for each task to be performed, let the participants perform the task from within the application, and recorded the results of the task. Specifically, when participants first started using the web application, the application provided

Screen#	*Kendall's Tau coeff.*	*Kendall's Tau p*	*Spearman coeff.*	*Spearman p*
1	0.6	0.142	0.8	0.104
3	1	-	1	-
5	0.733	0.039	0.829	0.042
6	1	-	1	-
7	1	-	1	-
8	1	-	1	-
9	1	-	1	-
10	0.714	0.024	0.857	0.014
11	0.8	0.05	0.9	0.037
12	1	-	1	-
13	1	-	1	-
14	1	-	1	-
18	0.867	0.015	0.943	0.005
20	1	-	1	-
21	0.8	0.05	0.9	0.037
22	0.733	0.039	0.829	0.042
23	0.6	0.142	0.7	0.188
25	1	-	1	-
26	1	-	1	-
28	0.667	0.174	0.8	0.2
29	1	-	1	-
30	1	-	1	-
32	1	-	1	-
34	0.8	0.05	0.9	0.037

Table 3: Correlation analysis results.

them with information on the purpose of the study. An example for each task was also shown to the participants to make sure they understood what they had to do.

To perform the tasks related to the relevance of the recommendations, participants were shown, for each screen of their sketch and in a random order, all the recommendations generated by GUIFetch for that screen. They were then asked to mark, using a checkbox, the recommendations they thought were irrelevant for that particular screen of their sketch.

To perform the tasks related to ranking the recommendations, rather than simply presenting all the recommendations sequentially and asking the participants to rank them, we asked for a set of pairwise comparisons. The rationale for this choice is that the number of recommendations depends on the participant's sketch and can be high. Due to the limitations of human short-term memory, remembering and comparing a potentially large number of recommendations shown in sequential order is difficult and error-prone. (We confirmed this issue in our pilot study, in which our participants were annoyed when they had to rank more than a few recommendations at a time.) With pairwise comparison, conversely, participants compare only two recommendations at a time, and we can still derive an overall ranking from the comparisons of the individual pairs.

After performing the above tasks, participants were asked whether they wanted to update their sketch based on GUIFetch's recommendations. If so, they could visualize the recommendations again while they were working on their sketch. We also explicitly mentioned to the participants to try as much as possible to avoid changes that they wanted to perform for reasons other than seeing the recommendations.

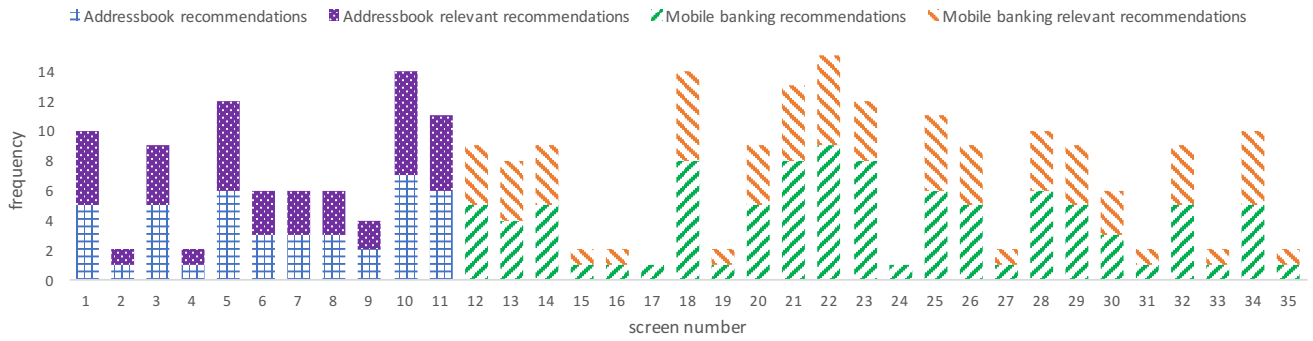

Figure 6: Number of (relevant) recommendations per screen.

Finally, we asked the participants to fill out a questionnaire about their background and their feedback on GUIFetch, based on their experience with it. In particular, the questionnaire asked participants about their experience with Android. It then asked them if GUIFetch's recommendations provided them with any insight or inspiration to update their initial sketch. If so, the questionnaire also asked which specific recommendations they used and why? Next, the questionnaire asked their opinions about the potential applications and extensions of GUIFetch. Finally, it gave them the option to provide any additional comments about their experience with GUIFetch.

4.3.3 GUIFetch Recommendations. Table 2 shows information about the sketches drew by participants and GUIFetch's recommendations for those sketches. The first column shows the participant id, followed by the name of the assigned category, the number of screens in the sketch, and the number of screens for which GUIFetch produced at least one recommendation. The number of sketch screens ranges from 1 to 7, with mean, median, and variance of 3.375, 3, and 2.65, respectively. The number of screens for which GUIFetch produced at least one recommendation ranges from 0 to 5, with mean, median, and variance of 2.1875, 2, and 1.3625, respectively. Note that both the mean and the total number of screens for the *address book* category are less than those for the *mobile banking* category, as *address book* apps generally provide less functionality compared to *mobile banking* apps. In both categories, the participants drew 18 and 36 screens, and GUIFetch was able to find recommendations for 11 (61%) and 24 (67%) of them, respectively. In total, GUIFetch produced at least one recommendation for 35 out of 54 screens (65%).

4.3.4 Relevance of GUIFetch Recommendations (RQ2). Figure 6 shows the total number of GUIFetch recommendations and the number of these recommendations that the users considered to be relevant. The results are presented for the 35 screens for which GUIFetch produced at least one recommendation.

As the figure shows, participants found the recommendations for 19 of the 35 screens (54%) to be 100% relevant. Additionally, the participants found 80% or more of the recommendations to be relevant for 8 screens (23%), and between 60% to 80% of the recommendations to be relevant for 6 screens (17%). For the remaining two screens, GUIFetch produced a single recommendation, which the participant found to be irrelevant. Overall, of the 138 recommendations provided by GUIFetch for 35 screens, participants found 112 recommendations (81%) to be relevant to their sketches. It is worth noting that 42 of the 138 recommendations are for the *address book* sketches, with 40 of them (95%) classified as relevant by the users, whereas the remaining 96 are for the *mobile banking* category, with 72 of them (75%) classified as relevant.

4.3.5 Matching of GUIFetch and User Ranking (RQ3). To assess the recommendations ranking provided by GUIFetch, we measured how well GUIFetch's ranking matched the participants' ranking using two well-established measures of non-parametric rank correlations: Kendall's tau [21] and Spearman's rho [41]. Kendall's tau is calculated based on the number of pairwise agreements between two ranking lists, and is computed using the formula $\frac{n_c - n_d}{N}$, where n_c is the number of concordant (ordered in the same way) pairs, n_d is the number of discordant (ordered differently) pairs, and N is the total number of pair combinations. Spearman's rho is calculated based on the difference in the ranking positions of each item in the ranking, and is computed using the formula $1 - \frac{6 \times \sum_1^n d_i^2}{n(n^2-1)}$, where d_i is the difference in paired ranks, and n is the number of items. We ran correlation analysis using both measures given the null hypothesis of no correlation between the rankings by GUIFetch and by the participants.

Table 3 shows the results of the correlation analysis. The first column in the table shows the screen number (same as figure 6), followed by Kendall's Tau's correlation coefficient and p-value, and Spearman's correlation coefficient and p-value. Out of the 24 screens for which GUIFetch produced more than one recommendation, and thus a ranking, user and GUIFetch rankings were identical for 14 screens (58%). For these rankings, the correlation coefficients for both measures are therefore one. For the remaining 10 screens (42%), Kendall's Tau's coefficients range from 0.6 to 0.867, while Spearman's coefficients range from 0.7 to 0.943. For 4 screens, the p-values for both Kendall's and Spearman's coefficients are less than 0.05, which means that the results are statistically significant, and we can reject the null hypothesis. For 3 screens, Spearman's p-values are less than 0.05 (i.e.,statistically significant), while Kendall's p-values equal 0.05, thus weakly rejecting the null hypothesis. For the remaining 3 screens, both Kendall's and Spearman's p-values are greater than 0.05, which means that, in these cases, there is not

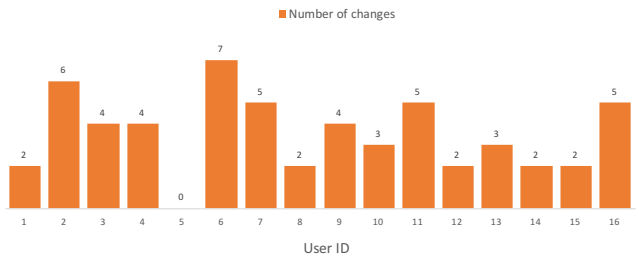

Figure 7: Number of sketch changes per participant.

enough evidence to reject the null hypothesis; we can therefore conclude that there is no correlation between the GUIFetch and user rankings for these 3 screens.

4.3.6 Insights from GUIFetch Recommendations (RQ4). To evaluate whether GUIFetch's recommendations provided any insights to participants on ways to update their initial sketches, we measured the number of changes they made to the sketches after seeing these recommendations. We also examined the feedback the participants provided in the questionnaire we asked them to complete.

We define a *change* as the action of adding, deleting, or updating any widget in the sketch. In the case of identical changes performed on multiple screens (e.g.,adding a sign-out button to all screens), we counted the change only once. In addition, we manually confirmed that every change performed was based on a recommendation from GUIFetch. Based on this definition, figure 7 shows the number of changes made to the sketches by each participant. As the figure shows, the number of changes ranges from 0 to 7, with mean, median, and variance of 3.5, 3.5, and 3.333, respectively. Out of 56 changes, 47 changes (84%) consisted of adding new widgets to the sketches, 5 changes consisted of updating an existing widget (9%), and 4 changes (7%) consisted of deleting a widget.

All participants but one—the same participant who also provided no recommendations in Table 2—answered "Yes" to the question of whether GUIFetch recommendations provided them with any new insight. Based on the answers in the questionnaire, we identified three common ways in which GUIFetch's recommendations helped the participants. (1) Reminders of basic, yet necessary GUI elements. Participant 10, for instance, added 3 GUI elements to his initial sketch and explained the change as follows: *"In a specific screen, I sometimes missed some essential components of the screen. So I could modify my screen design by reflecting GUIFETCH recommendations."* (2) Showing of design alternatives. Participant 3, for example, performed 4 updates to his initial sketch and elaborated: *"The fields in my screen were already similar to images X and Y. However, they were all text based. The image Z was a good visual representation of the app, and I just remembered the importance of images to represent information. Thus, in the refined design, I updated the text fields to icons"*. (3) Inspiring new ideas. Participant 13, for instance, added 3 GUI elements that were not in the recommendations, but were inspired by them, as the participant described: *"It helped with brainstorming and revising. It allowed me to see others' work, which helped to generate new ideas and missing aspects."*.

4.4 Discussion and Limitations

As the results of our experiments show, the participants do not find all the recommendations produced by GUIFetch to be relevant or useful. And the rankings provided by GUIFetch do not always match the user rankings. However, our results provide clear evidence that GUIFetch recommendations can be useful to users in a vast majority of cases. Besides, we speculate that, due to the somehow subjective nature of app matching, there is no one-size-fits-all possible ranking, and there may be an inherent upper bound on how accurate an approach can be. Despite this, in future work we plan to further analyze our results to see whether we can identify ways to further improve our technique.

To get a better understanding of possible applications and limitations of GUIFetch, we added to our questionnaire two follow-up questions on potential applications and extensions of GUIFetch and also gave the participants the ability to provide general feedback about the tool. A participant mentioned that GUIFetch might be used to do some quick design for throw-away, low-fidelity prototyping. Another participant thought that GUIFetch could serve as a useful checklist for checking the inclusion or exclusion of features and getting confidence that a consistent UI experience is provided to the user (akin to the concept of best practices). Yet another participant found GUIFetch useful in helping users think outside the box. Two other participants mentioned a new possible application for GUIFetch that we had not yet considered: using the tool for learning purposes. Specifically, one of them indicated that GUIFetch could help novice designers to learn more quickly, as it would allow them to compare their designs with the recommended ones, possibly finding flaws and improving the designs accordingly. Finally, one participant mentioned his experience taking an introductory mobile apps development course, in which the students struggled to find initial code to use as a starting point; he believed that having GUIFetch would have helped the students.

Regarding possible extension of GUIFetch, one of the participants mentioned that, since designers in digital space are dependent on applications such as Photoshop, Illustrator, Sketch, and InVision for designing mobile apps, it would be great if there were an existing database of what users are designing and can provide suggestions for a specific domain. Another participant proposed an add-on for current UI design toolkits that provides a checklist for essential GUI components when designing a screen. A participant also stated that it would be great to have a tool that checks the completeness of a sketch and gives recommendations.

The main drawback mentioned by the study participants is that they wanted more recommendations. As shown in the results section, GUIFetch was computed recommendations for 65% of the sketch screens, but it could not find any recommendation for the remaining 35% of the screens. Although the availability of the recommendations ultimately depends on which open source apps are available, we manually inspected the cases for which no recommendations were provided. We found that some of these sketch screens were not part of the essential functionality of their corresponding app category. Another reason we found has to do with drawing very ad-hoc screens. An address book sketch drawn by one of the participants, for instance, used so many personal names and images that GUIFetch was not able to find any match for it.

Another limitation, which we plan to address in future work, is that GUIFetch is currently unable to detect and compare images.

Besides the limitations mentioned by the participants, our initial experience shows that (unsurprisingly, in hindsight) it is easier to find good matches for types of apps that tend to provide a standard set of features, such as address books. Finding good matches for app categories that vary broadly (e.g.,fitness apps) or app categories that use many non-conventional GUI widgets (e.g.,games) tends to be more challenging. This seems to indicate that our approach may be more suitable for certain kinds of apps than for others.

5 RELATED WORK

Because GUIs are an essential part of apps, there has been a great deal of work on trying to assist developers in generating GUIs or automating this process.

5.1 GUI builders

Modern IDEs, such as Eclipse, Xcode, and Android Studio, provide users with GUI builders where users can drag and drop widgets and set the various properties of these widgets. Then, the corresponding code for the GUI is generated. Although this approach might simplify the process for developers, it does not provide enough flexibility to the users during early stages of GUI design, when the users need the freedom to sketch their ideas quickly. Besides, generating GUIs often happens as part of an iterative development approach, where GUIs are constantly updated or changed (e.g.,in agile development [13]). Also, the GUI code created by an actual developer is typically higher quality than the code created by an automated tool [49].

5.2 Generating GUI Code from Sketches

GUI sketches are employed in many companies during early stages of user interface design [24]. Some tools allow users to draw sketches and are able to recognize the GUI elements. JavaSketchIt [6] allows creating user interfaces through hand-drawn geometric shapes, identified by a gesture recognizer. SILK [25] allows designers to sketch a GUI using an electronic pad and stylus quickly; it then recognizes widgets and other interface elements as the designer draws them. de Sá and colleagues propose another prototyping tool that allows users to take a picture of a sketch and is then able to map the sketch elements to mobile widgets [10].

Other tools push this approach even further and not only recognize the GUI elements, but also generate code for the GUIs. MobiDev [40] provides users with a visual language of standard GUI elements that the users can use to draw app sketches; it then generates the apps based on these sketches. REMAUI [31] is another tool that infers GUI code for a mobile app from screenshots or conceptual drawings by using OCR and computer-vision techniques.

Although these tools can generate GUI code from sketches, they have limitations. MobiDev [40] requires users to use predefined GUI elements. REMAUI [31] only supports the top three Android widgets and is limited in generating code for a single GUI and does not support apps with multiple screens and transitions. Moreover, as we also stated above, the code generated automatically might not be as usable and reliable as code written and tested by users.

The work closest to ours is that of Reiss [38], which focuses on searching Java-based GUI code with a single screen. That technique is not directly applicable to mobile apps; creating a precise GUI model for apps is not straightforward because, as we explained in the Introduction, there are multiple ways to build app GUIs (i.e.,using XML layout resources, programmatically, or both). Therefore, building a precise model of an app GUI involves analyzing the app using a combination of static and dynamic analyses. Moreover, apps might consist of different screens and transitions. Therefore, it is essential to match the transitions between screens in addition to matching the screens themselves.

5.3 Code Search

Besides commercial code-search engines, such as GitHub (www.github.com) and OpenHub (code.openhub.net), there has been a large body of research on code search for helping developers find code in large public open source code repositories (e.g., [1, 4, 7, 8, 11, 15, 17–20, 26, 28–30, 34, 35, 39, 42–46, 48]). In this paper, we are particularly interested in searching Android apps and validating the GUIs of the apps against the user provided sketches. None of the existing code-search techniques could be directly used for this task, as they require code snippets or a single GUI as the starting point, rather than the sketch of an app.

5.4 Mining Web Designs

Data-driven approaches, such as Bricolage [23] and Webzeitgeist [22], mine a large number of designs and use heuristics to transfer the design of one web page to another. These techniques are different from our work, as they are in the context of web apps and use different approaches, such as knowledge discovery techniques.

6 CONCLUSION AND FUTURE WORK

We presented GUIFetch, a code-search technique that takes advantage of the growing number of open source apps in public repositories to provide users with code that can be used as a starting point for the apps they want to create. Given a sketch of an app (i.e., app's screens and transitions between them), GUIFetch searches for apps in public repositories that are as similar as possible to the provided sketch. The matching apps are then reported to the user, ranked by similarity to the sketch. GUIFetch can provide developers with a starting point for building their GUI-based apps, support early prototyping, and help designers assess whether there are existing apps similar to the one they want to develop. We implemented GUIFetch for Android apps and empirically evaluated it. Our results are promising, show that GUIFetch can help mobile apps' design and development, and motivate further research in this area.

We envision two main directions for future work, in addition to the ones we already discussed in the paper. First, we will perform additional studies involving more users and app types. Second, we will extend our technique so that it can also match non-GUI code, rank it, and report it to the users together with the GUI code. To do so, we plan to leverage earlier work [37]. Specifically, we will allow users to also specify input-output pairs and will add to GUIFetch the ability to generate glue code between non-GUI and GUI code.

7 ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grants CCF-1161821 and 1548856.

REFERENCES

[1] M. Akhin, N. Tillmann, M. Fahndrich, J. de Halleux, and M. Moskal. 2012. Search by example in TouchDevelop: Code search made easy. In *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*. IEEE, New York, 5–8.

[2] AndroidStudio. 2017. AndroidStudio. (2017). http://tools.android.com/build/studio.

[3] Mikhail J. Atallah and Susan Fox (Eds.). 1998. *Algorithms and Theory of Computation Handbook* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[4] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 681–682.

[5] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/1882291.1882316

[6] Anabela Caetano, Neri Goulart, Manuel Fonseca, and Joaquim Jorge. 2002. JavaSketchIt: Issues in Sketching the Look of User Interfaces. In *AAAI Spring Symposium on Sketch Understanding*. AAAI Press, Menlo Park, 9–14.

[7] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 10, 11 pages.

[8] Shih-Chien Chou, Jen-Yen Chen, and Chyan-Goei Chung. 1996. A Behavior-based Classification and Retrieval Technique for Object-oriented Specification Reuse. *Softw. Pract. Exper.* 26 (1996), 815–832.

[9] J.W. Creswell. 2013. *Educational Research: Planning, Conducting and Evaluating Quantitative and Qualitative Research*. Pearson, New Jersey. https://books.google.com/books?id=39LkngEACAAJ

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

[11] C. G. Drummond, D. Ionescu, and R. C. Holte. 2000. A learning agent that assists the browsing of software libraries. *IEEE Transactions on Software Engineering* 26, 12 (Dec 2000), 1179–1196.

[12] Espresso. 2017. Espresso. (2017). http://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[13] Harleen K. Flora, Swati V. Chande, and Xiaofeng Wang. 2010. Adopting an Agile Approach for the Development of Mobile Applications. (2010).

[14] GUIFetch. 2018. GUIFetch: Supporting App Design and Development through GUI Search. (2018). https://sites.google.com/view/guifetch/.

[15] R. J. Hall. 1993. Generalized behavior-based retrieval [from a software reuse library]. In *Proceedings of 1993 15th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 371–380.

[16] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 204–217.

[17] David Hemer and Peter Lindsay. 2002. Supporting Component-based Reuse in CARE. *Aust. Comput. Sci. Commun.* 24, 1 (Jan. 2002), 95–104.

[18] W. Janjic and C. Atkinson. 2012. Leveraging software search and reuse with automated software adaptation. In *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*. IEEE, New York, 23–26.

[19] W. Janjic, D. Stoll, P. Bostan, and C. Atkinson. 2009. Lowering the barrier to reuse through test-driven search. In *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, New York, 21–24.

[20] Jun-Jang Jeng and Betty H. C. Cheng. 1995. Specification Matching for Software Reuse: A Foundation. In *Proceedings of the 1995 Symposium on Software Reusability (SSR '95)*. ACM, New York, NY, USA, 97–105.

[21] M. G. Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 30, 1/2 (1938), 81–93. http://www.jstor.org/stable/2332226

[22] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092.

[23] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 2197–2206.

[24] James A. Landay and Brad A. Myers. 1995. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 43–50.

[25] J. A. Landay and B. A. Myers. 2001. Sketching interfaces: toward more human interface design. *Computer* 34, 3 (Mar 2001), 56–64.

[26] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. CodeGenie: Using Test-cases to Search and Reuse Source Code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 525–526.

[27] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in Android apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, 352–361.

[28] D. Lucredio, A. F. Prado, and E. S. de Almeida. 2004. A survey on software components search and retrieval. In *Proceedings. 30th Euromicro Conference, 2004.* IEEE, New York, 152–159.

[29] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering* 17, 8 (Aug 1991), 800–813.

[30] R. Mili, A. Mili, and R. T. Mittermeir. 1997. Storing and retrieving software components: a refinement based system. *IEEE Transactions on Software Engineering* 23, 7 (Jul 1997), 445–460.

[31] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 248–259.

[32] David Ortinau. 2016. From Sketch to App: Mobile Design to Development Workflow. (2016). http://rendr.io/from-sketch-to-app-a-design-to-development-workflow.

[33] Pencil. 2017. Pencil. (2017). http://pencil.evolus.vn.

[34] Andy Podgurski and Lynn Pierce. 1993. Retrieving Reusable Software by Sampling Behavior. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (July 1993), 286–303.

[35] T. Pole and W. Frakes. 1994. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering* 20 (08 1994), 617–630.

[36] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *j-jucs* 8, 11 (nov 2002), 1016–1038.

[37] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 243–253.

[38] Steven P. Reiss. 2014. Seeking the User Interface. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 103–114.

[39] Eugene J. Rollins and Jeannette M. Wing. 1991. Specifications as search keys for software libraries. In *Proceedings 8th International Conference on Logic Programming*. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 173–187.

[40] Julian Seifert, Bastian Pfleging, Elba del Carmen Valderrama Bahamóndez, Martin Hermes, Enrico Rukzio, and Albrecht Schmidt. 2011. 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)*. ACM, New York, NY, USA, 109–112.

[41] C Spearman. 2010. The proof and measurement of association between two things. *International Journal of Epidemiology* 39, 5 (2010), 1137–1150.

[42] J. Starke, C. Luce, and J. Sillito. 2009. Working with search results. In *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, New York, 53–56.

[43] Vijayan Sugumaran and Veda C. Storey. 2003. A Semantic-based Approach to Component Retrieval. *SIGMIS Database* 34, 3 (Aug. 2003), 8–24.

[44] Suresh Thummalapenta and Tao Xie. 2007. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proceedings ASE'07*. IEEE, New York, 204–213.

[45] F. Thung, S. Wang, D. Lo, and J. Lawall. 2013. Automatic recommendation of API methods from feature requests. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, 290–300.

[46] Taciana A. Vanderlei, Frederico A. Durão, Alexandre C. Martins, Vinicius C. Garcia, Eduardo S. Almeida, and Silvio R. de L. Meira. 2007. A Cooperative Classification Mechanism for Search and Retrieval Software Components. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*. ACM, New York, NY, USA, 866–871.

[47] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. 2015. Static Window Transition Graphs for Android (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, 658–668.

[48] Yunwen Ye and Gerhard Fischer. 2002. Supporting Reuse by Delivering Task-relevant and Personalized Information. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 513–523.

[49] Clemens Zeidler, Christof Lutteroth, Wolfgang Stuerzlinger, and Gerald Weber. 2013. *Evaluating Direct Manipulation Operations for Constraint-Based Layout*. Springer Berlin Heidelberg, Berlin, Heidelberg, 513–529.