

TOOLS FOR PROGRAM UNDERSTANDING AND REVERSE-ENGINEERING
OF MOBILE APPLICATIONS

by

TUAN ANH NGUYEN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2015

Copyright © by TUAN ANH NGUYEN 2015

All Rights Reserved

*To my family,
for their huge love and limitless support*

ACKNOWLEDGEMENTS

This thesis is inspired by the following people, who share my dream of discovering new methods to improve software programmers productivity specially in mobile.

I want to express my sincere appreciation to my supervising professor, Dr. Christoph Csallner for his support in my imagination and trust in my abilities. This final thesis paper would not have been possible without his advice and his unending patience. He has been a teacher, a colleague, and a friend helping me navigate around the many obstacles I encountered while attending UT Arlington. I am grateful to Dr. Manfrer Huber, Mr. David Levine and Dr. Matthew Wright for serving as members of my thesis committee.

This material is based upon work supported by the National Science Foundation under Grants No. 1017305 and 1117369.

Finally, I would like to thank my father, mother, sister, and my wife for their unbounded support. They are the pillars that give me strength and encouragement to help me through my time in school.

June 15, 2015

ABSTRACT

TOOLS FOR PROGRAM UNDERSTANDING AND REVERSE-ENGINEERING OF MOBILE APPLICATIONS

TUAN ANH NGUYEN, Ph.D.

The University of Texas at Arlington, 2015

Supervising Professor: Christoph Csallner

Mobile software development is evolving rapidly. Software development includes computer programming, documenting, testing and bug fixing processes. These processes need a detail understanding of the application logic which often requires reverse-engineering their artifacts. My thesis identifies and addresses the following three problems in mobile software development, specifically in program understanding and reverse-engineering for mobile application development. (1) There is no graphical on-phone debugger. (2) The second problem is that mobile software programmers have to manually re-implement the conceptual screen drawings or sketches of graphical artists in code, which is cumbersome and expensive. (3) Companies try to "go mobile" (by developing mobile apps). To do that understanding the high level business of their current legacy software systems is necessary but challenging.

To address these three challenges, this dissertation introduces the following three innovations. (1) GROPG is the first graphical on-phone debugger. GROPG makes debugging mobile apps more convenient and productive than existing text-based on-phone debuggers. (2) REMAUI is a mobile digital screenshot and sketch

reverse-engineering tool. REMAUI makes developing mobile user interface code easier. (3) RengLaDom is a legacy application reverse-engineering tool. RengLaDom can infer domain concepts from legacy source code.

Specifically, (1) debugging mobile phone applications is hard, as current debugging techniques either require multiple computing devices or do not support graphical debugging. To address this problem we present GROPG, the first graphical on-phone debugger. We implement GROPG for Android and perform a preliminary evaluation on third-party applications. Our experiments suggest that GROPG can lower the overall debugging time of a comparable text-based on-phone debugger by up to 2/3.

(2) Second, when developing the user interface code of a mobile application, a big gap exists between the sketches and digital conceptual drawings of graphic artists and working user interface code. Currently, programmers bridge this gap manually, by re-implementing the sketches and drawings in code, which is cumbersome and expensive. To bridge this gap, this dissertation introduces the first technique to automatically *reverse engineer mobile application user interfaces* from UI sketches, digital conceptual drawings, or screenshots (REMAUI). In our experiments on third party inputs, REMAUI's inferred runtime user interface hierarchies closely resembled the user interface runtime UI hierarchies of the applications that produced REMAUI's inputs. Further, the resulting screenshots closely resembled REMAUI's inputs and overall runtime was below one minute.

(3) Finally, a promising approach to understanding the business functions implemented by a large-scale legacy application is to reverse engineer the full application code with all its complications into a high-level abstraction such as a design document that can focus exclusively on important domain concepts. Although much progress has been made, we encountered the following two problems. (a) Existing techniques often cannot distinguish between code that carries interesting domain concepts and

code that merely provides low-level implementation services. (b) For an evaluation, given that design documents are typically not maintained throughout program development, how can we judge if the domain model inferred by a given technique is of a high quality? We address these problems by re-examining the notion of domain models in object-oriented development and encoding our understanding in a novel lightweight reverse engineering technique that pinpoints those program classes that likely carry domain concepts. We implement our techniques in a RengLaDom prototype tool for Java and compare how close our inferred domain models are to existing domain models. Given the lack of traditional domain models, we propose to use for such evaluation existing object-relational data persistence mappings (ORM), which map program classes to a relational database schema. The original application engineers carefully designed such mappings, consider them valuable, and maintain them as part of the application. After manually removing such OR mappings from open-source applications, our RengLaDom technique was able to reverse engineer domain models that are much closer to the original ORM domain models than the models produced by competing approaches, regardless of the particular ORM framework used. Additional experiments indicate that RengLaDom’s ability to infer better domain models extends to a variety of non-ORM applications.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
ABSTRACT	v
LIST OF ILLUSTRATIONS	xii
LIST OF TABLES	xv
Chapter	Page
1. INTRODUCTION	1
1.1 My Thesis	1
1.2 Problem 1: Debugging on a Phone is Difficult and There is No Graphical On-phone Debugger	2
1.3 Problem 2: Mobile Software Programmers Have to Manually Reimplement the Conceptual Screen Drawings or Sketches	3
1.4 Problem 3: Companies Try to "Go Mobile" so Understanding Current Legacy Software Systems is Necessary But Challenging	5
2. GROPG: A GRAPHICAL ON-PHONE DEBUGGER	7
2.1 Overview	7
2.2 Background on Android Debugging	8
2.2.1 Android Applications Are Signed Java Applications	8
2.2.2 Android's Infrastructure For Application Debugging	9
2.3 Graphical On-Phone Debugging	10
2.4 GROPG Implementation for Android Phones	12
2.5 Preliminary Experience	13
2.6 Related Work	15

2.7	Conclusions and Future Work	16
3.	REVERSE ENGINEERING MOBILE APPLICATION	
	USER INTERFACES WITH REMAUI	17
3.1	Introduction and Motivation	17
3.2	Motivating Example	19
3.3	Background	22
3.3.1	GUI View Hierarchy & Declarative GUI Programming	22
3.3.2	Example GUI Framework: Android	23
3.3.3	Optical Character Recognition (OCR)	24
3.4	REMAUI Overview and Design	25
3.4.1	Optical Character Recognition (Step 1)	26
3.4.2	Computer Vision (Step 2)	28
3.4.3	Merging (Step 3)	31
3.4.4	Identify Lists (Step 4)	33
3.4.5	Export (Step 5)	34
3.5	Research Questions	34
3.6	Evaluation	35
3.6.1	Subjects	36
3.6.2	RQ1: REMAUI Runtime	38
3.6.3	RQ2: Pixel-by-Pixel Similarity	39
3.6.4	RQ3: UI Hierarchy Similarity	40
3.7	Related Work	44
3.8	Threats to validity	46
3.8.1	Judging the quality of our tool output on concept drawings is subjective	46

3.8.2	The metrics we picked for comparing tool output to screenshots may not capture the “real” quality	47
3.8.3	We picked top-100 iOS apps. These may be biased towards simpler UI	47
3.9	Conclusions and Future Work	47
4.	REGLADOM: REVERSE ENGINEERING LEGACY APPLICATIONS INTO HIGH-LEVEL DOMAIN MODELS	49
4.1	Overview	49
4.2	Overview and Examples	51
4.2.1	Which are the domain model classes?	51
4.2.2	Motivating Example: Pizza Shop	58
4.3	Background	60
4.3.1	Analyzing only method bodies is enough to know about domain concept	60
4.3.2	Methods using classes	60
4.4	Implementation	62
4.4.1	Overview	62
4.4.2	Method Call Graph	64
4.4.3	Filtering Methods	65
4.5	Evaluation Methodology	69
4.5.1	Voting	69
4.5.2	Compare Analysis Results with Existing Object-Relational Mapping (ORM)	69
4.6	Evaluation	70
4.6.1	Subjects: A surprising result on non-database application	70

4.6.2	Subjects: Applications That Have An Object-Relational Mapping	71
4.7	Related Work	74
4.8	Conclusions and Future Work	77
	REFERENCES	78
	BIOGRAPHICAL STATEMENT	91

LIST OF ILLUSTRATIONS

Figure	Page
2.1 Architecture overview of a desktop-based debugger, Eclipse, using the Android debugging infrastructure for debugging an application running on a mobile phone.	9
2.2 DroidDebugger (left) and GROPG (right) while debugging the My Tracks application on a Samsung Galaxy S3 phone. In both cases the programmer set a breakpoint at line 580 of the MapFragment class. After My Tracks stopped at the breakpoint, the programmer stepped to line 581, inspected in-scope memory values, and updated the local variable named bottom.	11
2.3 The graphical on-phone debugger GROPG is built on the same ADB, JDWP, and JDI debugging infrastructure as the two-device debugger of Figure 2.1.	13
3.1 Example REMAUI use: The UI designer provides a conceptual UI drawing (left). REMAUI identifies UI elements such as lists of text and images and arranges them in a suitable UI hierarchy. REMAUI then exports the inferred UI as source code and resource files, compiles them, and runs them on a phone.	20
3.2 Example OCR performance at various granularity levels. Left to right: UI drawing and Tesseract-detected words, lines, blocks, and paragraphs.	22

3.3	Overview of REMAUI processing steps: (1) Locate and extract candidate words and lines with OCR; (2) locate and extract candidate UI elements as a hierarchy of nested bounding boxes using computer vision; (3) merge the results to improve recognition quality; (4) identify repeated items and summarize them as collections; (5) export the constructed UI as a mobile application for a given platform; (6) compile and execute.	25
3.4	Example results of the Table 3.1 heuristics: Input from Figure 3.3 (left), candidate words from OCR (framed, middle), and candidates eliminated by our heuristics (solid rectangles, right).	28
3.5	Computer vision processing steps from left to right: Original input bitmap; Edges detected via Canny’s algorithm as black and white; Dilated or broadened edges to swallow noise and join adjacent elements; Contours of the joined elements; Output: Hierarchy of the contours’ bounding boxes.	29
3.6	Example: Merging Figure 3.3 OCR lines with processed OCR words. . .	32
3.7	Runtime of REMAUI’s seven main processing steps on the 488 subjects, shown by group, from left (A) to right (E).	38
3.8	Normalized pixel-by-pixel screenshot similarity between REMAUI input and generated application on the 488 subjects, shown by group A–E from left to right. Higher values are better.	39
3.9	Part of a screenshot of Google Hangout (top), its UI hierarchy (middle), and the REMAUI-generated hierarchy (bottom). Each element is annotated at its center with its level in the UI hierarchy, with root=1. Each number’s color matches the color of its element’s boundary. . . .	41

3.10	Image, text, and overall UI element precision (p) and recall (r) for groups B (left) and C (right). Higher values are better.	43
4.1	RengLaDom is a three-step approach to automatic domain model reverse engineering. RengLaDom starts with identifying domain model classes. On those classes, it then identifies both basic and complex relations. (1) Extracting domain concept classes (2) Extracting domain model classes and their multiplicity basic relations (3) Ranking domain model classes classes relations	57
4.2	Entity-relationship diagram of Pizza Shop’s relational database. There are four entities, PizzaOrder, Topping, PizzaSize, and SysTime. There is an N-N relationship between PizzaOrder and Topping and an N-1 relationship between PizzaOrder and PizzaSize. SysTime does not have direct relationship with PizzaOrder, Topping, or PizzaSize but it is used to track time in days which links with <i>day</i> property in PizzaOrder. . .	59
4.3	Output of Womble tool in Pizza Shop project	60
4.4	All cases of a method access to objects and theirs fields to start navigating a business relation: (1) method foo directly accesses object of domain model class A and B, (2) method foo access object of domain model class A and directly calls a method bar that accesses a B domain object, (3) method foo access object of domain model class A and transitively calls a method bar that accesses a B domain object. . . .	62
4.5	Example of method call graph	65

LIST OF TABLES

Table		Page
2.1	Memory consumption of application and debugger [MB] and debugging time [min:s]. GROPG has a higher memory footprint but enables faster debugging.	14
2.2	GROPG needs fewer steps for setting a breakpoint.	15
3.1	Heuristics for eliminating likely false positive candidate words from the OCR results.	27
3.2	Heuristics for additional eliminations of OCR words, based on computer vision results.	31
4.1	Comparison Domain Model Class Relation defined by Hibernate Configuration and the ones reversed engineered by RengLaDom and Womble. With RengLaDom, we run it on both the original Pizza Shop application and on a version from which we stripped the Hibernate persistence. . .	61
4.2	Effectiveness of domain model class filtering between RengLaDom and Womble. Cl: Subject Java classes and interfaces. DC: Cl that are domain model classes. Rtr: Cl RengLaDom labeled as domain model classes. Pr: Precision. RC: Recall. RT: Time taken. Average: Average Pr and RC of Womble output. Intersection: Pr and RC of the intersection of Womble outputs. Union: Pr and RC of the union of Womble outputs. Group 1: projects without database. Group 2: projects using JDBC. Group 3: projects using iBATIS. Group 4: projects using MyBatis. Group 5: projects using Hibernate.	73

CHAPTER 1

INTRODUCTION

1.1 My Thesis

Mobile software development is evolving rapidly. My thesis identifies and addresses the following three problems in mobile software development, specifically in program understanding and reverse-engineering for mobile application development. (1) There is no graphical on-phone debugger. (2) The second problem is that mobile software programmers have to manually re-implement the conceptual screen drawings or sketches of graphical artists in code, which is cumbersome and expensive. (3) Companies try to "go mobile" (by developing mobile apps). To do that understanding the high level business of their current legacy software systems is necessary but challenging. Each of these three parts of my dissertation corresponds to one of the following three papers.

1. GROPG: A Graphical On-Phone Debugger (Proc. 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track, 2013)
2. Reverse Engineering Mobile Application User Interfaces With REMAUI (under review)
3. RengLaDom: Reverse Engineering Legacy Applications Into High-Level Domain Models (in preparation)

1.2 Problem 1: Debugging on a Phone is Difficult and There is No Graphical On-phone Debugger

Debugging is a common activity during software development and maintenance. For example, in a study of Java developers who were using the Eclipse IDE, over 90% of the developers used the built-in debugger to inspect memory values during program execution [1]. Likewise a recent study found professional software developers to use debuggers heavily for general program comprehension tasks [2].

Debugging mobile phone applications is hard, as current debugging techniques either do not provide powerful debugging features such as a graphical user interface or require multiple computing devices. Specifically, there are currently three types of debugging techniques for mobile phone applications. Mainstream are the first two, which run a standard debugger on a desktop computer. Type (1) techniques attach the debugger to a mobile phone emulator or virtual device that runs the debuggee application. All major mobile application platforms including Android, iOS, and Windows Phone provide such emulators. Emulators are useful but not sufficient, as they do not simulate all phone features precisely. Thus developers resort to type (2) techniques, which attach the desktop debugger to a phone, e.g., via a USB cable. The mainstream techniques therefore ultimately require two connected computing devices, which we also call the *two-device requirement*.

Type (3) techniques were recently pioneered by TouchDevelop [3] and DroidDebugger¹. These techniques only require a single device, i.e., a mobile phone. However these techniques do not provide the powerful features of desktop-based debuggers. That is, type (3) techniques are either text-based (DroidDebugger) or lack basic debugging features (TouchDevelop). Overall, missing in type (3) techniques is a user interface that allows the programmer to (a) quickly navigate and manipulate the de-

¹<https://play.google.com/store/apps/details?id=net.sf.droiddebugger>

buggee memory and (b) view the debuggee’s current source code location, memory values, call stack, and user interface side-by-side.

The two-device requirement of mainstream debugging excludes many, e.g., those who own a smartphone but cannot afford a second computing device such as a desktop computer. This requirement also limits people who have multiple devices, as it constrains debugging style. For example, to get real location sensor readings, a developer has to move the phone, which can be a hassle if the phone is connected to a desktop computer with a short USB cable.

When abandoning mainstream debugging, developers are confronted with the lack of powerful features of on-phone debugging. From the history of debugging we already know that certain debugger features enable higher debugging efficiency. For example, consider the well-known tradeoffs between textual vs. graphical debuggers exemplified by GDB and DDD [4, 5]. We thus argue that mobile phone application debugging overall could be significantly improved by adding powerful features to on-phone debuggers. We present our initial work on providing on-phone debugging with a powerful graphical user interface.

1.3 Problem 2: Mobile Software Programmers Have to Manually Reimplement the Conceptual Screen Drawings or Sketches

Developing the user interface code of mobile applications is cumbersome and expensive in practice. Due to the early consumer and entertainment focus of the two major platforms Android and iOS and the high competitive pressure in the mobile application market, users have come to expect mobile user interfaces that are highly customized and optimized for the task at hand [6, 7]. To satisfy this demand, mobile user interfaces often deviate from their platforms’ standard user interface (UI)

components and provide their own novel or customized UI elements such as buttons, dividers, and custom element positioning and grouping.

To create such optimized user interfaces, the development process of mobile applications routinely incorporates non-programmers. User experience (UX) designers and graphic artists design, customize, and optimize each screen of the user interface with a mix of prototyping techniques. Common prototyping techniques include paper-and-pencil and pixel-based concept drawings created in Photoshop or similar graphic design tools [8, 9, 6, 10].

Our key observation is that there is a gap in the production process, as user interface concept drawings have to be converted into working user interface code. Currently, these conversions are done manually by programmers, which is cumbersome, error-prone, and expensive. While modern IDEs such as Eclipse, Xcode, and Android Studio have powerful interactive builders for graphical user interface (GUI) code [11, 12], using such a GUI builder to re-create a complex user interface drawing is a complex task. For example, in an evaluation of GUI builders on a set of small tasks, subjects using Apple’s Xcode GUI builder introduced many bugs that later had to be corrected. Subjects produced these bugs even though the study’s target layouts were much simpler than those commonly found in third-party mobile applications [12].

This challenge is compounded in practice. (1) First, custom layouts are often desired but it is harder to create them with a stock GUI builder. (2) Second, the conversion from user interface concept drawing to user interface code is typically performed many times during an application’s lifespan. The reason is that many development teams follow an iterative approach, in which a user interface may undergo many revision during both initial software development and maintenance.

This gap in the mobile application development process is significant as many mobile applications are being developed and maintained. For example, In the USA

over 90% of consumers over 16 years of age use a mobile phone and more than half of the mobile phones are smartphones, mostly running Android or iOS [13]. On these smartphones, people use mobile applications to perform many tasks that have traditionally been performed on desktop computers [14, 15, 13, 16]. Example tasks include reading and writing emails, listening to music, watching movies, reading the news, and consuming and producing social media. To date, more than one million mobile applications have been released². Automating the conversion from user interface design drawings to working user interface code may therefore save a lot of time and money, which could be put to better use.

1.4 Problem 3: Companies Try to "Go Mobile" so Understanding Current Legacy Software Systems is Necessary But Challenging

Reverse engineering is a key software engineering task [17, 18]. Reverse engineering is needed to maintain or increase the value of an existing piece of software. Reverse engineering infers high-level models of the software, such as domain models, that allow a software engineer to gain some insight in and an understanding of a piece of code. Only this understanding allows the engineer to then perform crucial maintenance tasks such as fixing bugs, refactoring the code, adapting the software to requirements that have changed, or adding new functionality.

Reverse engineering today often means that engineers are given the source code of a C++, Java, or C# application they are not familiar with. Besides this unfamiliar source code, these reverse engineers are often not given additional design models or documents that may be helpful for their task. This situation is caused by the original, forward, engineers and architects, who do not like to document their design

²<http://www.appbrain.com/stats/number-of-android-apps>

models [19] and do not keep design documents up to date or in synch with the source code [20].

Recent studies have shown that having a domain model in a relational style such as an UML class diagram can improve software maintenance tasks [21, 22, 23]. Taken together, a promising approach is to reverse engineer object-oriented programs into relational domain models such as UML class diagrams. Parts of this task have been automated in reverse engineering tools, such as Womble [24] and class2uml [25].

Although much progress has been made, we encountered the following two problems. (a) Existing techniques often cannot distinguish between code that carries interesting domain concepts and code that merely provides low-level implementation services. (b) For an evaluation, given that design documents are typically not maintained throughout program development, how can we judge if the domain model inferred by a given technique is of a high quality?

We implement our techniques in a RengLaDom prototype tool for Java and compare how close our inferred domain models are to existing domain models. Given the lack of traditional domain models, we propose to use for such evaluation existing object-relational data persistence mappings (ORM), which map program classes to a relational database schema. Our RengLaDom technique was able to reverse engineer domain models that are much closer to the original ORM domain models than the models produced by competing approaches, regardless of the particular ORM framework used. Additional experiments indicate that RengLaDom’s ability to infer better domain models extends to a variety of non-ORM applications.

CHAPTER 2

GROPG: A GRAPHICAL ON-PHONE DEBUGGER

2.1 Overview

The most significant challenge of adding powerful features to on-phone debuggers is limited screen real estate, as dictated by the comparatively small mobile phone screen size. From desktop-based debugging however, developers are used to seeing during debugging several kinds of information on the screen, including the debugged program's current source code location in the context of the surrounding code, the program's runtime memory values, call stack, and user interface.

Besides screen size there are other issues that make it hard to port an existing desktop debugger to mobile phones. Desktop debuggers use user interface elements optimized for keyboard short-cuts and mouse interaction, which do not exist on phones. Similarly, the user interface libraries desktop debuggers are constructed from typically do not exist on phones. For example, Android does not provide the SWT Standard Widget Toolkit on which the mainstream Java and Android debugger Eclipse is built.

Powerful on-phone debugging has only recently come within reach, with mobile phones evolving to powerful interactive computers, narrowing the performance gap with desktop computers in terms of CPU and memory resources. More importantly, mobile phones only recently started to provide high-resolution touch-screens, which enable interactive debugging.

In this chapter we present our initial work on providing on-phone debugging with a powerful graphical user interface. That is, we describe the design and im-

plementation of GROPG, the first **G**raphical **O**n-**P**hone Debugger. We implement GROPG for Android as Android is a mobile phone platform that is used widely. We also describe a preliminary comparison of our GROPG prototype with the most closely related tool, the text-based on-phone debugger DroidDebugger.

Although we describe our work in terms of Android application debugging, our techniques can also be applied to debugging related languages on related mobile phone platforms such as iOS and Windows Phone. Our implementation is open source and freely available¹.

2.2 Background on Android Debugging

In this section we provide necessary background information on the Android operating system and its Java-like infrastructure for application debugging.

2.2.1 Android Applications Are Signed Java Applications

Android is an open source operating system that is used widely for mobile phones. For example, it is estimated that in the second quarter of 2012 various vendors shipped a total of over 100 million Android phones². Android consists of operating system services, a Linux-based kernel, and the Dalvik virtual machine. Dalvik is conceptually similar to a Java virtual machine and provides similar memory safety.

While Android is mostly implemented in C, almost all Android applications are written in Java. An application is compiled from Java or Java bytecode to the Dalvik Executable (DEX) bytecode language, the language Dalvik can execute. Android runs an application only if it is digitally signed. Developers can compile and sign

¹<http://cseweb.uta.edu/~tuan/GROPG/>

²<http://www.idc.com/getdoc.jsp?containerId=prUS23638712>

an application either for release or for debugging. If an application is compiled for release mode, Android prevents it from being debugged.

2.2.2 Android’s Infrastructure For Application Debugging

For application-level debugging, Dalvik implements two interfaces defined by the Java virtual machine. This makes it easy for existing Java debuggers to connect to Dalvik and control debuggee applications. At a high level, these two interfaces follow the cooperative debugging model, which assumes that the virtual machine is implemented correctly and faithfully provides its debugging infrastructure. Cooperative debugging is standard for application debugging. Specifically, these two interfaces are the Java Debug Interface JDI and the Java Debug Wire Protocol JDWP and are part of the Java Platform Debugger Architecture JPDA.

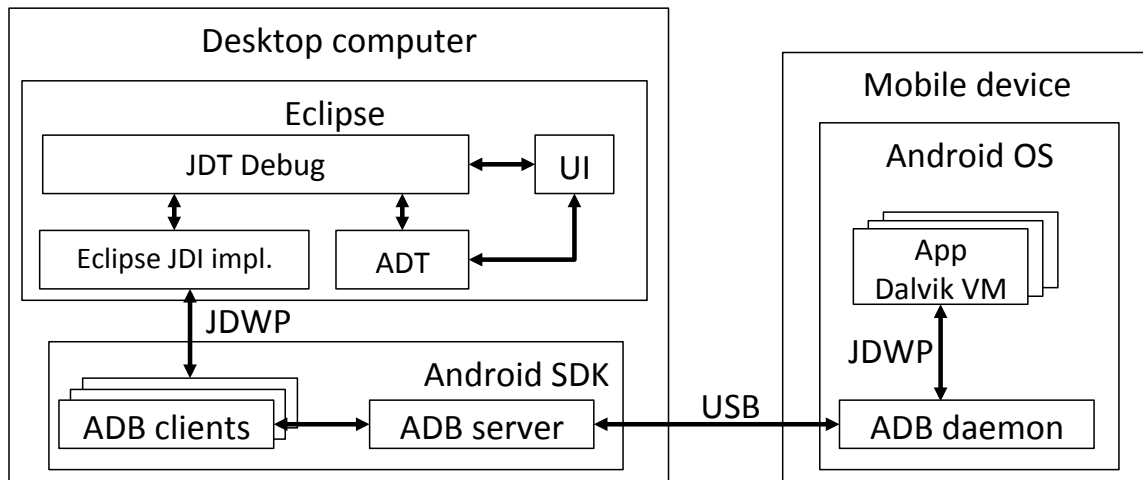


Figure 2.1: Architecture overview of a desktop-based debugger, Eclipse, using the Android debugging infrastructure for debugging an application running on a mobile phone.

A debugger can communicate with Dalvik via the Android Debug Bridge ADB, which consists of client, server, and daemon. Each Android device runs a daemon,

which may be discovered by the server. The server can be discovered by clients, which are typically started by tools such as a debugger.

Figure 2.1 illustrates the Android debugging infrastructure in the context of two-device debugging. The Eclipse IDE is connected with an Android application running on a mobile phone via a communication stack consisting of, bottom-up, a USB cable, the Android Debug Bridge ADB, the Java Debug Wire Protocol JDWP, and the Java Debug Interface JDI. Since Android follows the standard Java debugging interfaces JDWP and JDI, the Eclipse Java debugger only needs minimal adaptation for Android debugging. This adaptation is provided by the Android Development Tools ADT, which direct debugger interactions over the Android Debug Bridge.

2.3 Graphical On-Phone Debugging

The biggest challenge in the design of a graphical on-phone debugger is the design of its user interface, as the user interface has to provide within a mobile phone's constraints the features developers expect from desktop-based debugging.

Figure 2.2 shows the main user interface of our GROPG graphical on-phone debugger on the right and compares it with the closest related tool, the text-based on-phone debugger DroidDebugger. At a high level, GROPG and DroidDebugger make available similar kinds of information.

The main difference between the two approaches is in how they display and let the user interact with debugging information. DroidDebugger provides a text-based user interface, essentially via a shell. The user types command lines and DroidDebugger prints its responses as text output. GROPG on the other hand provides information graphically and interactively. For example, whereas DroidDebugger just prints out the fields of an object, GROPG users can expand such fields and their child fields iteratively by tapping on the desired fields in a graphical display of such a

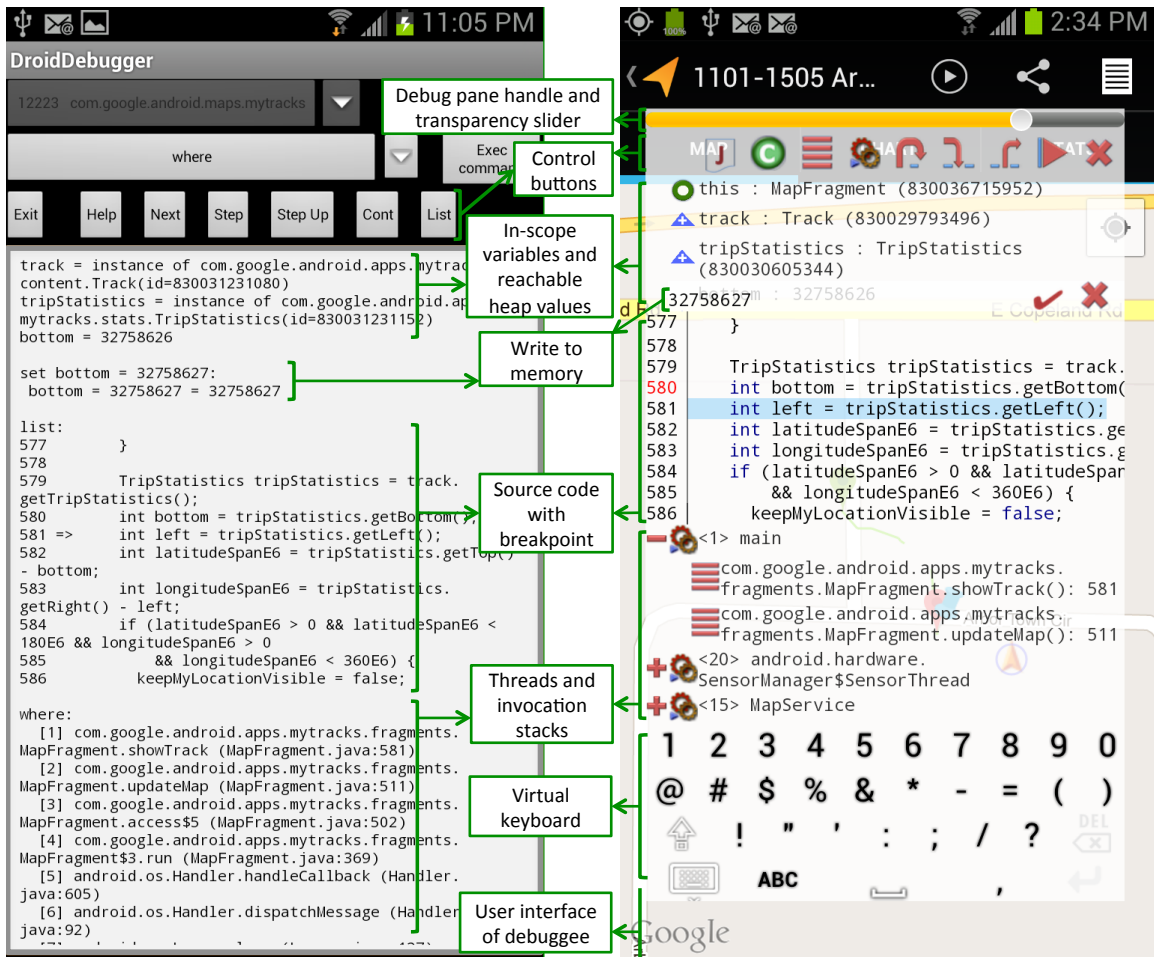


Figure 2.2: DroidDebugger (left) and GROPG (right) while debugging the My Tracks application on a Samsung Galaxy S3 phone. In both cases the programmer set a breakpoint at line 580 of the MapFragment class. After My Tracks stopped at the breakpoint, the programmer stepped to line 581, inspected in-scope memory values, and updated the local variable named bottom.

linked data structure. Moreover, whereas DroidDebugger occupies the entire screen, GROPG displays the debugger in a transparent layer on top of the debuggee application.

If the developer wants to interact with the debugged application's user interface, DroidDebugger forces her to frequently switch contexts between debuggee and debugger, as the debugger occupies the entire screen. GROPG on the other hand

provides a transparent layer or debug pane, which allows the user to interact with debugger and debuggee on the same screen, saving the user from frequent context switches. The user can move the debug pane by dragging its handle and adjust the pane's transparency with its slider, to provide just enough visual contrast between debugger and the underlying debuggee user interface.

GROPG enables a workflow that mirrors debugging of a desktop application. That is, the user can load source code files into the debugger, navigate through them, set breakpoints by tapping a line, view and edit the list of all active breakpoints, inspect in-scope memory values, step into, over, and out of instructions, inspect the current threads and runtime stacks, jump to calling methods, and change the values of in-scope memory and heap locations. All actions are available via graphical on-phone interactions, via tap and multi-touch.

2.4 GROPG Implementation for Android Phones

Figure 2.3 shows an overview of the GROPG implementation for Android. We re-use many components of the two-device setup of Figure 2.1. That is, debugger and Dalvik communicate via ADB and the standard debugging interfaces JDWP and JDI. But instead of a USB cable, our ADB components communicate directly over a network socket. External machines should not connect to this socket and take control of our virtual machine with debug commands. We prevent that scenario by only allowing local connections to this socket. This on-phone communication via a network socket works for Android versions 2.3 to 4.2.1.

To run all components on a single mobile phone, we structure our system as a graphical front-end of the minimal debugger JDB. This setup mirrors the architecture of DroidDebugger, which builds on another JDB Android port. JDB in turn is built

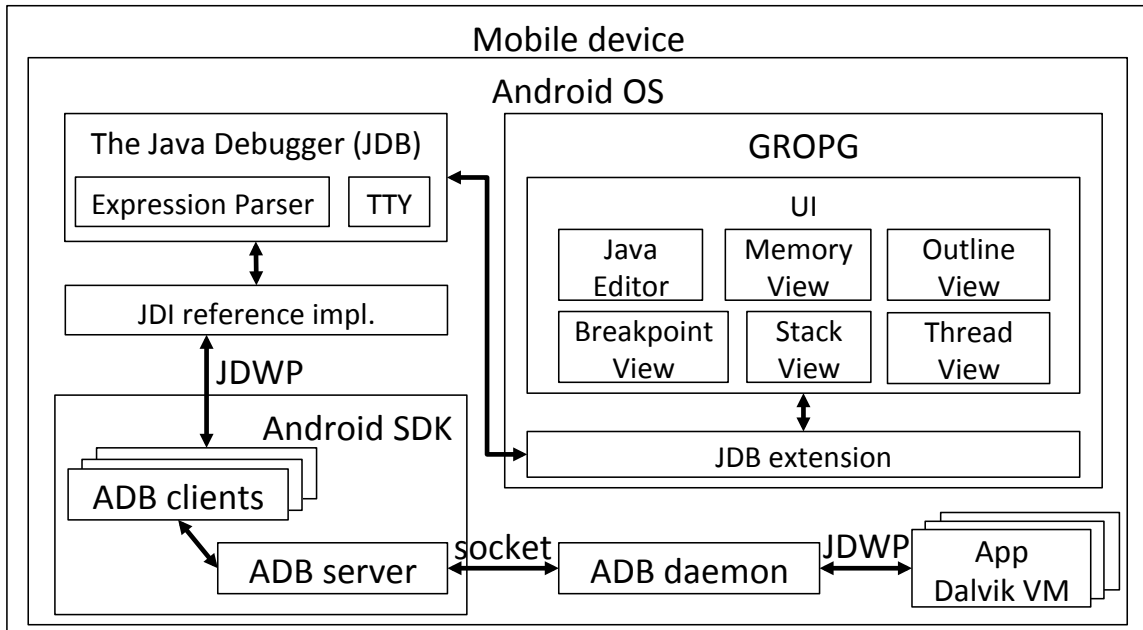


Figure 2.3: The graphical on-phone debugger GROPG is built on the same ADB, JDWP, and JDI debugging infrastructure as the two-device debugger of Figure 2.1.

on the JDI reference implementation and is one of the example debuggers that are part of JDPa.

Following is the main workflow. After setting a breakpoint and starting the debuggee, the programmer interacts with the full debuggee UI. When the debuggee is paused (e.g., at a breakpoint), GROPG displays its debug pane on top of the debuggee UI. After performing debug actions, the programmer switches back to interacting with the full debuggee UI. At this point GROPG saves the current state of the debug pane and removes it from the screen. GROPG will recreate its debug pane UI state at the next debuggee execution break.

2.5 Preliminary Experience

We compare GROPG with the closest related tool, DroidDebugger, on a typical debugging task performed on three example open source applications. Table 2.1

summarizes the applications and their size. F2C prompts the user for a temperature and converts it from Fahrenheit to Celsius. My Tracks³ uses a phone’s GPS sensor to record the user’s motion outdoors. ZXing⁴ is a library that enables barcode scanning via a phone’s camera. All measurements were taken on a dual-core 1.5GHz, 2GB RAM Samsung Galaxy S3 mobile phone running the recent Android 4.0.4 version.

Table 2.1: Memory consumption of application and debugger [MB] and debugging time [min:s]. GROPG has a higher memory footprint but enables faster debugging.

Debuggee	LOC	DroidDebugger			GROPG		
F2C	27	3	10	2:13	3	24	45
My Tracks	21,563	19	11	3:35	19	26	52
ZXing	5,756	7	12	2:53	7	25	1:01


The debugging task we chose for this comparison is to (1) start debugger and debuggee, (2) attach debugger to debuggee, (3) set one breakpoint, (4) once the debuggee reaches the breakpoint step to the next instruction, and (5) display current memory values and the frame stack of the current thread. We set the breakpoints at F2C class F2CActivity line 17, My Tracks class MapFragment line 580 (Figure 2.2), and ZXing class CaptureActivity line 440. For each task, Table 2.1 shows the peak memory consumption and the total time spent.

For space reasons we can break out sub-step measurements only for one step. Table 2.2 shows the sub-steps of step (3), setting a breakpoint, for My Tracks. The individual times depend on user experience and will vary among users. But the bottom line is that a GROPG user can set a breakpoint faster and with fewer sub-steps, as she can leverage GROPG’s comprehensive debug pane and graphical user interface.

³<https://play.google.com/store/apps/details?id=com.google.android.maps.mytracks>

⁴<https://play.google.com/store/apps/details?id=com.google.zxing.client.android>

Table 2.2: GROPG needs fewer steps for setting a breakpoint.

	DroidDebugger		GROPG	
1	Open source file in external tool	21	Open source file	18
2	Review code in external tool	19	Review code	19
3	Remember class, line for breakpoint	5		
4	Switch back to debugger	3		
5	Recall command syntax	0		
6	Type <i>stop at com.google.android.apps. my-tracks.fragments.MapFragment:580</i>	63	Tap line	1
7	Tap <i>Exec command</i>	1	Tap  button	1
		112		39

Given the well-known trade-offs between graphical and text-based debugging our measurements are not surprising. GROPG has a higher memory overhead, due to the addition of a graphical front-end. The memory overhead is modest, with some 25MB on a 2GB RAM phone, and does not seem to increase significantly with debuggee size. Table 2.1 also suggests that GROPG can reduce debugging time by up to 2/3.

2.6 Related Work

People have written graphical front-ends for text-based debuggers before. A prominent example is the DDD graphical Data Display Debugger front-end for GDB [4, 26]. Another example is the C/C++ Eclipse-plugin CDT, which also implements a graphical front-end for GDB [5].

DDD has a console window that allows GDB-style textual debugger control. However none of the graphical debuggers are designed to run on a mobile device with a small screen such as a smart phone.

Existing powerful graphical Android debuggers cannot be easily ported to Android. For example, the Eclipse-Java development tools, a popular PC graphical Java

debugger, designed specifically for platform, cannot be ported to mobile device. Its user interface (Standard Widget Toolkit) has to rebuild from scratch. We takes advantage of mobile input features such as touch and gesture and build GROPG as a new full feature mobile graphics debugger. GROPG also is optimized for low memory and CPU requirements on mobile devices.

Losely related is recent work on powerful on-phone IDEs, i.e., TouchDevelop [3]. TouchDevelop does not have a debugger and does not have to deal with a large existing code base that was written on desktop computers, as TouchDevelop defines its own Pascal-like language and is written and primarily displayed on phone screens.

There are some Android-based IDEs that can run on mobile phones such as AIDE. However none of these existing IDEs has a debugger and they are also mostly targeted at tablets that have an external keyboard.

In a different context there has been initial work on debugging a rule-set on a mobile phone, which has to address similar screen real-estate issues [27]. However, the notion of a rule-set is different from debugging a complex running Android application.

2.7 Conclusions and Future Work

We described GROPG, the first graphical on-phone debugger. For future work, we want to integrate our approach with on-phone coding [3], to create a full on-phone IDE. We also want to address many of the traditional debugging challenges on-phone, including cross-language Java and native code debugging [28] and support for why-questions [29].

CHAPTER 3

REVERSE ENGINEERING MOBILE APPLICATION

USER INTERFACES WITH REMAUI

3.1 Introduction and Motivation

Converting a conceptual drawing of a screen into good user interface code is hard, as it is essentially a reverse engineering task. As in other reverse engineering tasks, general principles have to be inferred from specific instances. For example, a suitable hierarchy of user interface elements has to be inferred from a flat set of concrete pixels.

Compared to other reverse engineering tasks such as inferring design documents from code [17, 30, 31, 32, 33], an unusual additional challenge is that the input, i.e., the pixels, may originate from scanned handwriting and human sketches with all their imperfections [34, 35, 36]. This means that sets of pixels have to be grouped together and recognized heuristically as images or text. Then groups of similar images and text have to be recognized heuristically as example elements of collections. And for the UI of innovative mobile applications, at each step the recognized elements may diverge significantly from the platform's standard UI elements.

For professional application development, one may wonder if this reverse engineering step is artificial. That is, why are meaning and source code hierarchy of screen elements not explicitly encoded in the conceptual design drawings if these are done in digital tools such as Photoshop? One reason is that some UX designers start with pencil on paper, so it would be desirable to convert such drawings directly into working user interface code.

More significantly, when UX designers create digital bitmap images (typically by drawing them in Photoshop), the digital design tools do not capture the hierarchy information that is needed by user interface code. More importantly, it is not clear if UX designers and graphic artists want to think in terms of source code hierarchies.

While this gap is most apparent in forward engineering, there may also exist a traditional reverse engineering scenario. A developer may only have access to screenshots of a mobile application, maybe after losing all other software artifacts such as the source code. In such a situation it would be desirable to automatically infer from the screenshots the user interface portion of the missing source code.

This paper therefore identifies and addresses three problems in mobile application development. In reverse engineering, we address the problem of inferring the user interface code of a mobile application from screenshots. In forward engineering, we address the gap between scanned pencil-on-paper UI sketches and code as well as the gap between pixel-based UI sketches and code. While these problems occur at different times in the development process, they share the task of pixels-to-code inference.

Specifically, this paper introduces the first technique to automatically *Reverse Engineer Mobile Application User Interfaces* (REMAUI). REMAUI automatically infers the user interface portion of the source code of a mobile application from screenshots or conceptual drawings of the user interface. On a given input bitmap REMAUI identifies user interface elements such as images, text, containers, and lists, via computer vision and optical character recognition (OCR) techniques. REMAUI further infers a suitable user interface hierarchy and exports the results as source code that is ready for compilation and execution. The generated user interface closely mimics the user interface of a corresponding real application. To summarize, the paper makes the following major contributions.

- The paper describes REMAUI, the first technique for inferring mobile application user interface code from screenshots or conceptual drawings.
- To evaluate REMAUI, we implemented a prototype tool that generates the UI portion of Android applications. This tool is freely available via the REMAUI web site.
- In an evaluation on 488 screenshots of over 100 popular third-party mobile applications, REMAUI-generated UIs were similar to the originals, pixel-by-pixel and in their runtime UI hierarchy.

3.2 Motivating Example

As a motivating example, assume a UX designer has produced the screen design bitmap shown in the left of Figure 3.1. The top of the screen contains the user’s profile image and an icon. Below is a list, in which each entry has a person’s image on the left, the person’s name and text message in the middle, and the message date on the right. List entries are separated by horizontal bars. The bottom of the screen has four icons and their labels.

REMAUI infers from this bitmap working UI code, by mimicking the steps a programmer would take. REMAUI thus uses vision and character recognition techniques to reason about the screen bitmap. REMAUI groups related pixels into text or images, lines of text into text boxes, related items into containers, and repeated elements into list elements. REMAUI thus identifies non-standard user interface components such as arbitrarily shaped items (e.g., the round images on the left) and non-standard lists (e.g., using the special horizontal separator).

```
<RelativeLayout <!-- List Entry ... --> >
  <ImageView <!-- Horizontal Bar ... --> />
  <ImageView android:id="@+id/ImageView_1"
    android:layout_width="59dip"
    android:layout_height="61dip"
```

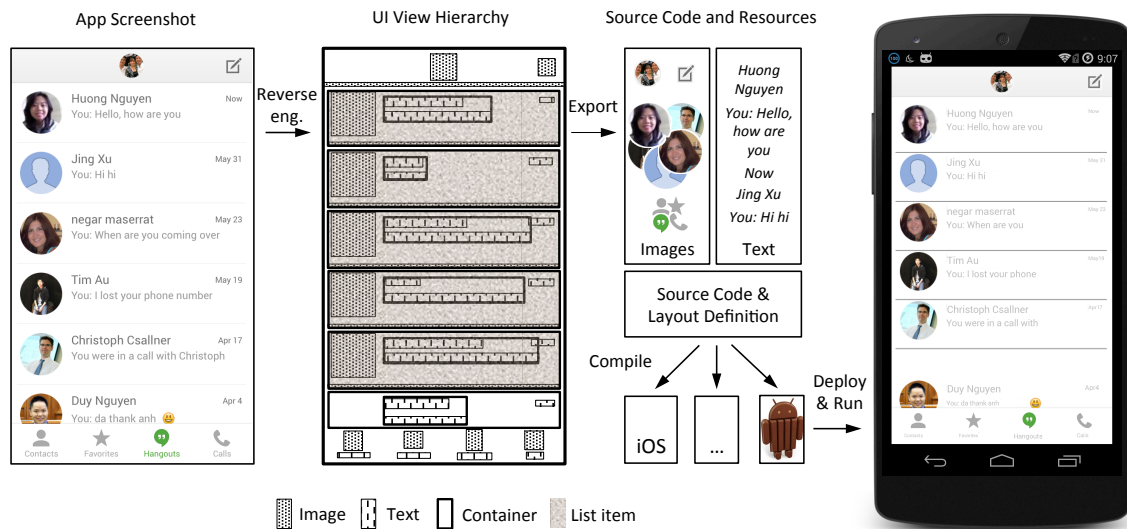


Figure 3.1: Example REMAUI use: The UI designer provides a conceptual UI drawing (left). REMAUI identifies UI elements such as lists of text and images and arranges them in a suitable UI hierarchy. REMAUI then exports the inferred UI as source code and resource files, compiles them, and runs them on a phone.

```

android:layout_marginLeft="5dip"
android:layout_marginTop="0dip"
android:src="@drawable/img_9"
android:scaleType="fitXY"
android:layout_alignParentTop="true"
android:layout_alignParentLeft="true" />
<RelativeLayout <!-- Nested: Text Block (center) ... --> >
  <TextView <!-- Sender name ... --> />
  <TextView <!-- Message ... --> />
</RelativeLayout>
<TextView <!-- Message date (right) ... --> />
</RelativeLayout>

```

Listing 3.1: REMAUI-generated layout for each list entry of Figure 3.1. Details are only shown for the left part of a list entry.

REMAUI generates several XML files to capture the screen’s static properties. In our example, the main XML file declares and positions the elements of the top and bottom rows including icons and their labels. This file also contains a list view for the bulk of the screen content. The layout of each list entry is defined by the Listing 3.1 XML file. For example, it positions a contact’s image and aligns it with the top left

of its parent (*alignParentTop*, *alignParentLeft*). REMAUI recognizes aligned text blocks such as the sender's name and message, groups them into a (nested) layout container (Listing 3.1), and exports the recognized text fragments as an Android resource file. At application runtime the list entries are added by the also generated Listing 3.2 Java source code.

```
public class MainActivity extends Activity {  
    //..  
    private void addListView0() {  
        ListView v = (ListView) findViewById(R.id.ListView_0);  
        final ArrayList<ListI> values = new ArrayList<ListI>();  
        values.add(new ListI(R.drawable.img_4, R.drawable.img_9, R.string.string_0, R.string.  
            string_1, R.string.string_2));  
        //..  
    }  
}  
//..
```

Listing 3.2: REMAUI-generated Android (i.e., Java) source code that populates Listing 3.1 list entries at application runtime.

The generated UI code and layout definitions can be compiled with standard Android development tools. Moreover, the code is similar to how a professional developer would implement the screen. For example, the generated code uses the appropriate kinds of layout container such as `RelativeLayout` for the list entries. A `RelativeLayout` can eliminate the need for some nested containers and thus keep the layout hierarchy relatively flat, which improves rendering performance at application runtime.

3.3 Background

This section contains necessary background information on GUI programming, modern mobile phone GUIs, and computer vision and optical character recognition (OCR).

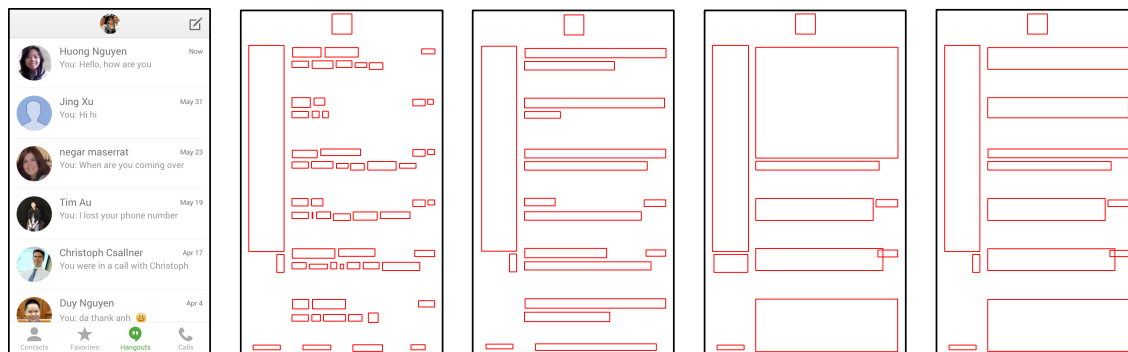


Figure 3.2: Example OCR performance at various granularity levels. Left to right: UI drawing and Tesseract-detected words, lines, blocks, and paragraphs.

3.3.1 GUI View Hierarchy & Declarative GUI Programming

The graphical user interface (GUI) of many modern desktop and mobile platforms is structured as a view hierarchy [37, 38]. Such a hierarchy has two types of nodes, leaf nodes (images, buttons, text, etc.) and container nodes. The root view represents an application’s entire space on screen. The root can have many transitive children. Each child typically occupies a rectangular sub-region of its parent. Each view can have its own parameters such as height, width, background color, and position. A view can be positioned relative to the root or other views such as its parent or siblings.

Mobile platforms such as Android and iOS render a view hierarchy on screen in-order. This means a parent is drawn before its children and a child hides parts of

its parent. Siblings are drawn in the order they are defined. A best practice is to minimize rendering time waste by keeping hierarchies flat and avoiding view overlap.

Given the relatively small mobile phone screen size, mobile platforms make it easy to hide their default screen elements such as the iOS title bar or the Android navigation bar. Applications often use this feature to maximize screen size.

To define basic GUI aspects, modern platforms provide two alternatives. The traditional desktop approach is construction through regular program code [37]. The now widely recommended alternative is declarative [38, 39, 40], e.g., via XML layout definition files in Android. Advanced GUI aspects are then defined programmatically, which typically leads to a combination of code and layout declaration files.

Building an appealing user interface is hard [41, 37]. Besides understanding user needs, the GUI facilities of modern platforms are complex and offer many similar concepts to choose from. This challenge is especially significant for developers new to their target platform. While each platform provides standard documentation and sample code, these samples often produce unappealing results.

3.3.2 Example GUI Framework: Android

The Android standard libraries define various GUI containers (“layout containers”) and leaf nodes (“widgets”). According to an August 2012 survey of the 400 most popular non-game applications in the Google Play app store [42], the following containers were used most frequently: `LinearLayout` (130 uses per application on average) places its children in a single row or column; `RelativeLayout` (47) positions children relative to itself or each other; `FrameLayout` (15) typically has a single child; `ScrollView` (9) is a scrollable `FrameLayout`; and `ListView` (7) lays out children as a vertical scrollable list.

The following widgets were used most frequently: `TextView` (141) is read-only text; `ImageView` (62) is a bitmap; `Button` (37) is a device-specific text button; `View` (17) is a generic view; `EditText` (12) is editable text; and `ImageButton` (11) is a device-independent button that shows an image. Besides the above, the Android library documentation currently lists some additional two dozen widgets and some three dozen layout containers.

3.3.3 Optical Character Recognition (OCR)

To infer UI code that closely reproduces the input conceptual drawing, RE-MAUI distinguishes text from images and captures the text as precisely as possible. Decades of research into optical character recognition (OCR) have produced specialized methods for recognizing various kinds of text such as text in different sizes, fonts, and orientation, as well as handwritten text [34, 35]. Generally it is easier to recognize text online (while it is being written) than offline. Similarly, it is easier to recognize print than handwriting.

Existing OCR tools perform relatively well if the input consists of mostly text. A good example is single-column text with few images. Current OCR tools perform worse if the text density is lower and text is arranged more freely and combined with images [43]. A good representative OCR tool is the powerful and widely used open-source OCR engine Tesseract [44, 45], which, for instance, Mathematica 9 uses to recognize text. In the closely related task of segmenting pages (for example, to distinguish images and individual text columns), Tesseract performs on par with commercial tools [45, 46].

However, the limitations of such a powerful OCR tool on complex inputs become apparent when subjecting it to screenshots or conceptual UI drawings. For example, Figure 3.2 shows from left to right a conceptual drawing and Tesseract’s results when

detecting text at various granularity levels, i.e., words, lines, blocks, and paragraphs. In this example, Tesseract finds all words but also classifies as words non-words such as the contacts' images. In general, for the domain of conceptual screen drawings and screenshots Tesseract's precision and recall are often both below one in all granularity levels. So even a powerful OCR tool may miss some words and classify non-text as words.

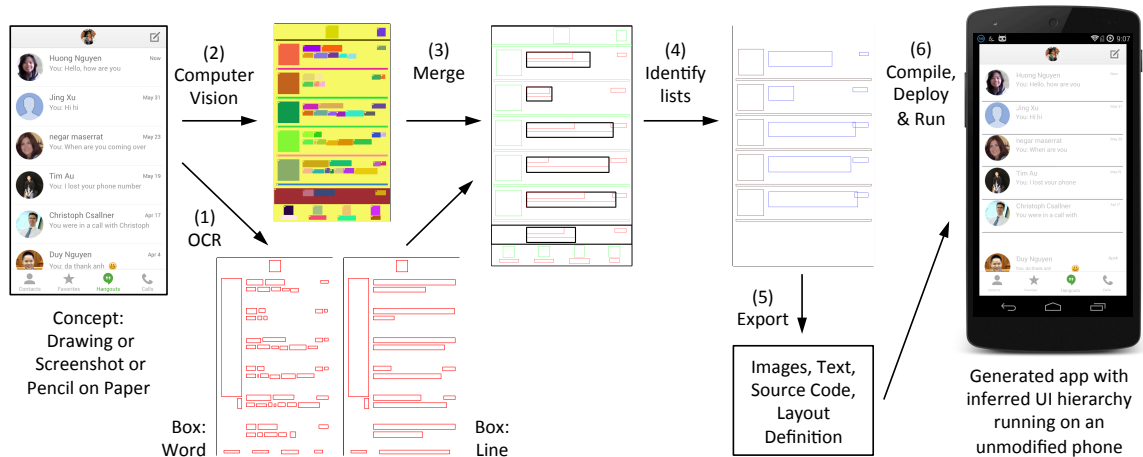


Figure 3.3: Overview of REMAUI processing steps: (1) Locate and extract candidate words and lines with OCR; (2) locate and extract candidate UI elements as a hierarchy of nested bounding boxes using computer vision; (3) merge the results to improve recognition quality; (4) identify repeated items and summarize them as collections; (5) export the constructed UI as a mobile application for a given platform; (6) compile and execute.

3.4 REMAUI Overview and Design

Figure 3.3 shows REMAUI's six main processing steps. At a high level, these steps are carried out by REMAUI's three main components. The core component is a powerful off-the-shelf optical character recognition (OCR) engine (step 1). Since OCR produces false positive candidate words, REMAUI filters the OCR results with

its domain-specific heuristics. Both to further compensate for OCR’s limitations and to identify non-text elements such as images, REMAUI combines OCR with a powerful off-the-shelf computer vision system (step 2).

In two-dimensional images computer vision techniques can quickly detect features such as corners and edges. Computer vision has therefore been applied to diverse tasks such as recognizing faces in two-dimensional images of the world or to allow self-driving cars to detect the edge of the road [47]. Using computer vision REMAUI approximates the boundaries of each screen element such as text and images.

In its final component, REMAUI merges OCR and computer vision results (step 3) and in the merged data identifies structures such as lists (step 4). REMAUI then exports the inferred user interface as a combination of layout declarations and program source code for the given target mobile platform (step 5), compiles this combination to binaries, and runs the binaries on an unmodified smartphone (step 6).

Not shown in Figure 3.3 is a pre-processing step in which REMAUI removes standard operating system title and navigation bars, if they are present. Since these screen areas are standardized it is relatively easy to detect and remove them.

3.4.1 Optical Character Recognition (Step 1)

First, REMAUI applies on the given input bitmap off-the-shelf OCR word detection. Since optical character recognition suffers from false positives, REMAUI post-processes OCR results to remove candidate words that likely do not reflect true words in the input. Figure 3.4 visualizes this process on the example bitmap from Figure 3.3. At word-level detection, REMAUI’s OCR system classifies several UI elements as a word that are not a word but an image or a part of an image.

To remove likely false positive words, REMAUI encodes knowledge about its mobile phone UI domain as heuristics, summarized in Table 3.1. As an example, rule 3 encodes that on a phone screen a word is likely not cut off and thus does not extend beyond the border of the screen. This rule is specific to phone screens and does not apply in all the settings the off-the-shelf OCR engine may be applied in outside REMAUI.

Table 3.1: Heuristics for eliminating likely false positive candidate words from the OCR results.

#	Name	Heuristic
1	Zero	$h = 0 \vee w = 0$
2	Long	$w/h < 0.05 \vee h/w < 0.05$
3	Cut off	$x < 0 \vee y < 0 \vee x + w > W \vee y + h > H$
4	Conf.	$c \leq 0.4$
5	Content	$c \leq 0.7 \wedge (\frac{ e_h/e_w - h/w }{\max(e_h/e_w, h/w)} > 0.5 \vee \frac{ a-e }{\max(a,e)} > 0.8)$
6	No-text	$[\backslash\text{p}\{\text{C}\}\backslash\text{s}]^* \vee [\text{^\}\backslash\text{x}00\text{-}\backslash\text{x}7\text{F}]^*$

The heuristics are given in terms of the input data, the OCR results, and heuristic values computed by REMAUI. Specifically, from the input UI screen available are its width (W) and height (H). The OCR system produces for each of its candidate words the word’s height (h), width (w), area ($a = w * h$), font family and size, upper left corner coordinates (x, y), text content (t), and confidence level (c). The confidence level is derived from the distance of the word’s characters from idealized characters [48].

From the text content and font information produced by OCR for a given word, REMAUI estimates the width (e_w), height (e_h), and area (e) the candidate word should occupy given the font size and family. Rule 5 uses this information to remove a word if, within bounds, the text area estimated by REMAUI does not match

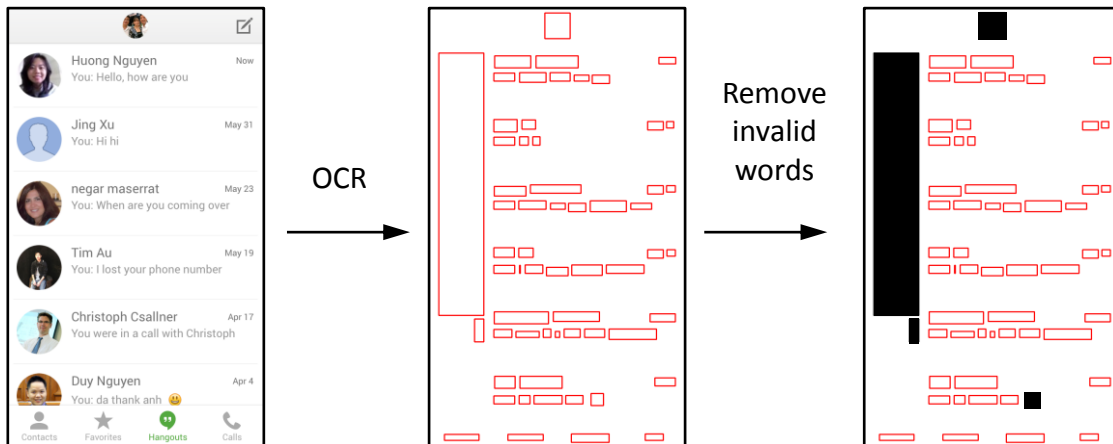


Figure 3.4: Example results of the Table 3.1 heuristics: Input from Figure 3.3 (left), candidate words from OCR (framed, middle), and candidates eliminated by our heuristics (solid rectangles, right).

the text area reported by OCR. This rule removed all four candidate words that are removed in the right side of Figure 3.4.

The other rules exclude words OCR is not confident about (rule 4), have a zero dimension (rule 1), or have an odd shape (rule 2). An odd shape likely does not capture an English-language word, as they are long and narrow, vertically or horizontally. Finally, rule 6 removes words that only contain non-ASCII characters or only consist of control characters and whitespace.

The heuristics’ constants are derived through trial and error on a small set of third-party bitmaps. The resulting heuristics have held up reasonably well on the much larger set of third-party bitmaps used in the evaluation (Section 3.6).

3.4.2 Computer Vision (Step 2)

In this step REMAUI infers a first candidate view hierarchy. Two important observations are that (1) many vastly different view hierarchies can lead to very similar if not identical on-screen appearances and (2) a programmer will likely find some of

these view hierarchies more valuable than others. REMAUI therefore follows carefully chosen heuristics to produce desirable view hierarchies that balance the following two goals.

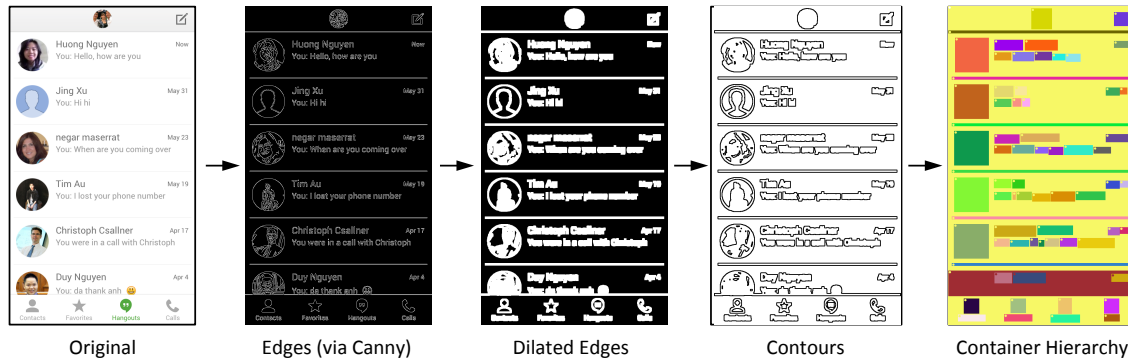


Figure 3.5: Computer vision processing steps from left to right: Original input bitmap; Edges detected via Canny’s algorithm as black and white; Dilated or broadened edges to swallow noise and join adjacent elements; Contours of the joined elements; Output: Hierarchy of the contours’ bounding boxes.

The first goal is a minimal hierarchy, i.e., having a minimum number of nodes. From the programmer’s perspective this is important to prevent clutter in the generated code. More importantly, drawing a large number of views slows down the application. For example, a programmer would not want a container that contains one child view for each character of every word displayed by the container.

However, a competing goal is maximum flexibility of the inferred view hierarchy. Distinct UI elements should be represented by distinct views to allow the generated UI to be well displayed on various combinations of screen size and resolution. Thus, a programmer would, for instance, not want to represent the four distinct buttons of the Figure 3.3 bottom-screen navigation bar as a single image. However, combining these four buttons into a single image and a single leaf view would reduce the number of views.

To infer a good candidate view hierarchy, REMAUI first tries to identify all atomic visual elements in the input UI. By atomic we mean a visual element that reasonably should not be divided further. For example, an icon is atomic but so can be an entire text paragraph. For each identified atomic visual element REMAUI then computes its approximate view.

To achieve these tasks, REMAUI leverages off-the-shelf computer vision. Figure 3.5 illustrates REMAUI’s key computer vision steps on the Figure 3.3 example input bitmap. First we detect the edges of each image element via Canny’s widely used algorithm [49, 47]. But these edges themselves are not good candidates for atomic elements as, for example, each character or even minor noise would become its own element.

To merge close-by elements with each other and with surrounding noise and to close almost-closed contours REMAUI dilates its detected edges. REMAUI uses a heuristic to, for example, allow a word’s characters to merge but keep words separate. REMAUI then computes the dilated edges’ contours. Each contour is a candidate atomic element.

Figure 3.5 also illustrates the heuristic nature of this process. The last list entry shown in the input screen is cut off by a horizontal dividing line. Edge detection, dilation, and contour thus all merge the last list item with the dividing line, reducing REMAUI’s precision and recall of atomic visual elements.

Finally, REMAUI computes the bounding box of each candidate atomic element, to approximate the element’s view. Recall from Section 3.3.1 that typically each view is rectangular and fully contained in its parent. Partially overlapping boxes are thus merged into a new bounding box. A fully contained box becomes the child view of the containing box.

3.4.3 Merging (Step 3)

In two sub-steps REMAUI merges the results of OCR and computer vision to heuristically combine the best aspects of both and to integrate the OCR-inferred text into the vision-inferred candidate view hierarchy.

First, REMAUI removes OCR-detected words that conflict with vision-inferred element bounding boxes. This step addresses common OCR false positives such as classifying part of an image as a text fragment, classifying bullet points as “o” or a similar character, and merging lines of text that have too little spacing. The resulting OCR-extracted text is not useful and should instead be exported as an image.

Table 3.2: Heuristics for additional eliminations of OCR words, based on computer vision results.

#	Description
1	Word aligns vertically & overlapped $\geq 70\%$ with ≥ 2 vision boxes that do not overlap each other
2	Word aligns horizontally & overlapped $\geq 70\%$ with ≥ 2 vision boxes, distance between each pair of boxes $>$ each box’s size
3	Word contains a non-leaf vision box
4	Word contains only 1 vision box, box size < 0.2 word size
5	Non-overlapped leaf vision box contains only 1 word, word size < 0.2 box size
6	If leaf vision box’s words are $> 50\%$ invalidated, invalidate the rest
7	If > 3 words are the same text and size, aligned left, right, top, or bottom, each has < 0.9 confidence, and are non-dictionary words
8	Leaf vision box contains a word, $M < 0.4 \vee (M < 0.7 \wedge m < 0.4) \vee (M \geq 0.7 \wedge m < 0.2)$, with $m = \min(\frac{w}{b_w}, \frac{h}{b_h})$, $M = \max(\frac{w}{b_w}, \frac{h}{b_h})$

Specifically, each OCR word is subjected to the Table 3.2 heuristics. In addition to the OCR word’s width (w) and height (h), we now also have the computer vision bounding box’s width (b_w) and height (b_h). For example, rule (1) checks if an OCR

word overlaps with two vision boxes whose y-coordinates do not overlap. This happens if OCR merged two text lines whereas the vision results kept them separate.

REMAUI further removes OCR words that are not contained by an OCR line (using the OCR lines from step 1). REMAUI then merges OCR words and lines into text blocks. OCR lines often blend together into a single line unrelated text that just happened to be printed on the same line. For example, the Figure 3.3 contact names (left) appear on the same line as message dates (right). However they are conceptually separate. REMAUI thus splits a line if the word-level OCR indicates that the distance between two words exceeds a heuristic threshold (i.e., their height). Figure 3.6 shows this process for the Figure 3.3 example. REMAUI adds the resulting text blocks to the view hierarchy and removes the vision boxes they overlap with.

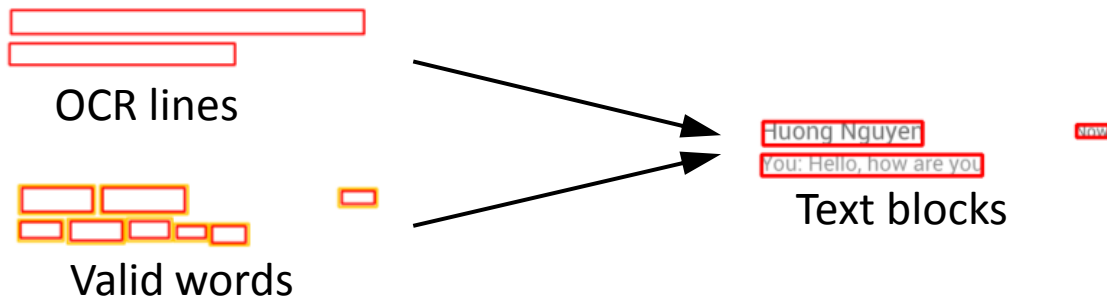


Figure 3.6: Example: Merging Figure 3.3 OCR lines with processed OCR words.

REMAUI aims at extracting text contents with high precision. The employed OCR engine produces better text contents when treating its input as a single text line. This way the OCR engine does not have to reason about which parts of the input are text versus non-text. REMAUI thus invokes OCR on each text block (in line mode), yielding text that resembles the text in the input relatively closely. Finally,

REMAUI groups close-by text blocks into a container, if the vertical distance between text blocks is less than either of their heights.

3.4.4 Identify Lists (Step 4)

In this step REMAUI identifies repeated items and summarizes them as collections, for two reasons. First, the final UI definition is more compact and efficient if each repeated resource is only represented once. Second, this step allows REMAUI to generalize from a few instances to a generic collection. REMAUI can then supply the observed instances as an example instantiation of the collection.

REMAUI identifies repeated instances by ordering the views by their relative location and searching them for identical sub-trees. A sub-tree consists of a view and a subset of its transitive children. Two sub-trees are identical if each of their child nodes has a peer in the other sub-tree, such that both nodes have the same number of children and the same width, height, type (text or image), and matching location within its parent (each within a threshold). Neither text contents nor image bitmaps have to be identical, as a list item may, for example, contain the face of a user as in Figure 3.3.

If identical sub-trees are found, REMAUI creates a bounding box around each of them. Each box contained in such a bounding box that is not part of the sub-tree belongs to the list item anchored by the sub-tree. However, such an overlapping box varies across list elements and will be exported as an optional element of the list entry. The properties of these optional elements are determined by overlaying all of them and using the resulting bounding boxes.

3.4.5 Export (Step 5)

In this step REMAUI exports all results as an Android project directory, complete with relevant source code and resource files. This directory can be compiled with standard Android IDEs. Specifically, REMAUI crops and extracts each identified image from the input screenshot, only once for repeated images. To provide a reasonable background color, REMAUI uses as the container’s background the dominant color of each container after extracting all identified images. REMAUI exports all detected text content and format to Android *strings.xml* and *styles.xml* files. REMAUI exports layout files to the Android *layout* directory, for the layout shared between list entries and for the main screen. Finally, REMAUI generates Java code to fill lists with the identified entries at runtime.

3.5 Research Questions

To evaluate REMAUI, we ask (a) if it is currently feasible to integrate REMAUI into a standard mobile application development setup and (b) if REMAUI-generated user interfaces are useful in the sense that the generated UI is similar to the UI an expert developer would produce. We therefore investigate the following three research questions (RQ), expectations (E), and hypotheses (H).

- RQ1: What is REMAUI’s runtime in a standard development setup?
 - E1: Given its expensive OCR and computer vision techniques, we do not expect REMAUI to run interactively.
 - H1: REMAUI can run on a standard development machine in a similar amount of time as a software installation wizard, which we approximate as up to one minute.

- RQ2: Is a REMAUI-generated UI visually (pixel by pixel) similar to a given third-party input UI conceptual drawing?
 - E2: Given their wide variety, we do not expect REMAUI to work well for all applications.
 - H2: REMAUI produces a UI that is visually similar to an input UI conceptual drawing, when running on non-game mobile applications.
- RQ3: Is the view hierarchy of a REMAUI-generated UI similar to the view hierarchy of a given third-party input application?
 - E3: Given their wide variety, we do not expect REMAUI to work well for all applications.
 - H3: REMAUI produces a UI whose view hierarchy is similar to the view hierarchy of a given handwritten non-game mobile application.

3.6 Evaluation

To explore our research questions we implemented REMAUI for Android. Our prototype generates Android code and resource files that are ready to be compiled and executed. Our prototype supports, among others, Android’s three most popular layout containers and three most popular widgets (Section 3.3.2). For off-the-shelf OCR, REMAUI uses the open source engine Tesseract [44] and Tesseract’s default version 3.0.2 English language data trained model. This means that REMAUI currently does not use Tesseract’s options for training its classifiers. Step 1 uses Tesseract’s fastest mode¹ with fully automatic page segmentation. For off-the-shelf computer vision, REMAUI uses the open source engine OpenCV [36] in its default configuration, without training.

¹TESSERACT_ONLY, PSM_AUTO

For the evaluation, REMAUI ran on a 16 GB RAM 2.6 GHz Core i7 MacBook Pro running OS X 10.10.2. Android experiments were run on a 2 GB RAM Google Nexus 5 phone running Android 4.4.4. Additional iOS experiments were run on a 1 GB RAM iPhone 5 phone running iOS 7.1.2.

3.6.1 Subjects

Using existing third-party applications to explore our research questions is a good fit for several reasons. (1) First, it is straightforward to capture a screenshot of a running application and hand such a screenshot to REMAUI. It is also straightforward to compare such screenshots pixel by pixel with REMAUI-generated screenshots (RQ2). (2) More importantly, having a running application enables inspecting the application’s UI hierarchy. We can then compare this hierarchy with the corresponding UI hierarchy of the REMAUI-generated application (RQ3).

The high-level workflow of our experiment is as follows. We first ran a subject Android or iOS application on a corresponding phone. At some point we took a screenshot and at the same time captured the current UI hierarchy. We (only) handed the captured screenshot to REMAUI and thus obtained a generated application. We then ran the generated application on an Android phone and, at the same time, took a screenshot and captured the runtime hierarchy. To clarify, no UI hierarchy information was provided to REMAUI.

Obtaining the UI hierarchy at runtime required low-level OS access. We thus used a rooted Nexus 5 for Android and a jail-broken iPhone 5. To obtain the view hierarchy on Android we used Android’s `uiautomator`² via the Android Debug Bridge [50] (`adb shell uiautomator dump`). For iOS, we used `cycrypt` [51] recursively starting from the root view.

²<http://developer.android.com/tools/help/uiautomator>

A REMAUI-generated application’s aspect ratio (between output screen width and height) is the same as the one of its input screenshot. With this aspect ratio, a REMAUI-generated application supports many screen resolutions, via Android’s standard density-independent pixel (dp) scheme. The Android runtime thereby scales a REMAUI-generated application’s dp units based on a device’s actual screen density.

Since our REMAUI prototype is implemented for Android, our first group of subjects consists of third-party non-game Android applications. To sample popular applications, we downloaded the top-100 free Android applications from the Google Play store as of November 9, 2014. From these we excluded games, as most games do not provide GUIs through a view hierarchy but through the native OpenGL library. This left us with 46 top-100 applications, covering (except games) all application categories present in the top-100, such as e-commerce, email, maps, media players, productivity tools, translation software, and social media. The REMAUI web site lists name and version of each subject application used in the evaluation. From each application, we captured the application’s main screen (in the form it appears after starting the application). We refer to these subjects as group C.

To broaden our set of subjects, and since many developers first target iOS, we added iOS applications. We downloaded on August 12, 2014 the top 100 free iOS applications from the Apple App Store. The resulting 66 non-game top-100 applications cover a range of categories similar to group C. We took a screenshot of every screen we could reach, yielding 302 screenshots (group A). For each application we took another screenshot showing the main screen with different data contents, yielding 66 subjects (group B). Since iOS 7 defined a new design language and Google and Apple are major application developers, we included from them 58 screenshots of iOS 7 applications outside the top-100 (group D).

There may also be a use case of manually drawing designs and scanning them. Since such third-party drawings are hard to obtain, we created sketches of 16 screenshots (group E). These screenshots are our manual renderings of the main screen of the alphabetically first 16 of the top 100 iOS applications in the Apple app store as of August 12, 2014.

3.6.2 RQ1: REMAUI Runtime

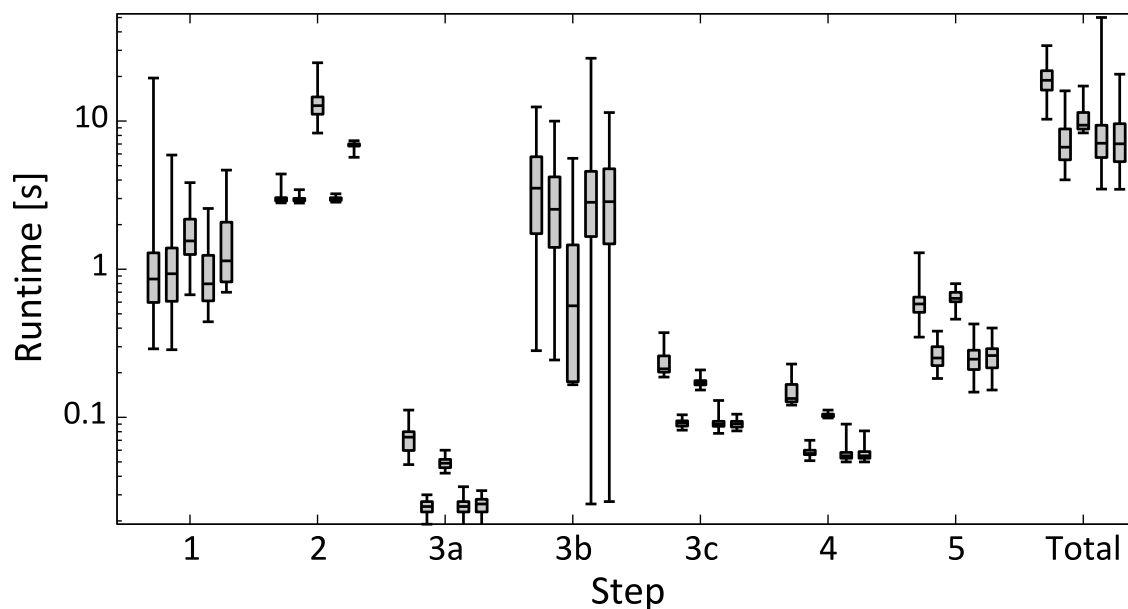


Figure 3.7: Runtime of REMAUI’s seven main processing steps on the 488 subjects, shown by group, from left (A) to right (E).

Figure 3.7 shows the runtime of REMAUI’s seven major processing steps, i.e., (1) OCR, (2) computer vision, (3a) merging OCR text with vision boxes, (3b) splitting text lines, (3c) creating the view hierarchy, (4) identifying lists, (5) export, and total runtime. Each step shows the runtimes of groups A–E from left to right. Not surprisingly, steps 1, 2, and 3b took longest, as these are the only steps that call

REMAUI’s computer vision and OCR engines. The cost of step 3b varied widely, as the number of OCR calls depends on the number of identified text blocks. Step 5 includes extracting a bitmap for each image view, which also takes time. On a modern desktop computer total runtime was well within the one minute time frame, with a 52 second maximum and an average total runtime of 9 seconds.

3.6.3 RQ2: Pixel-by-Pixel Similarity

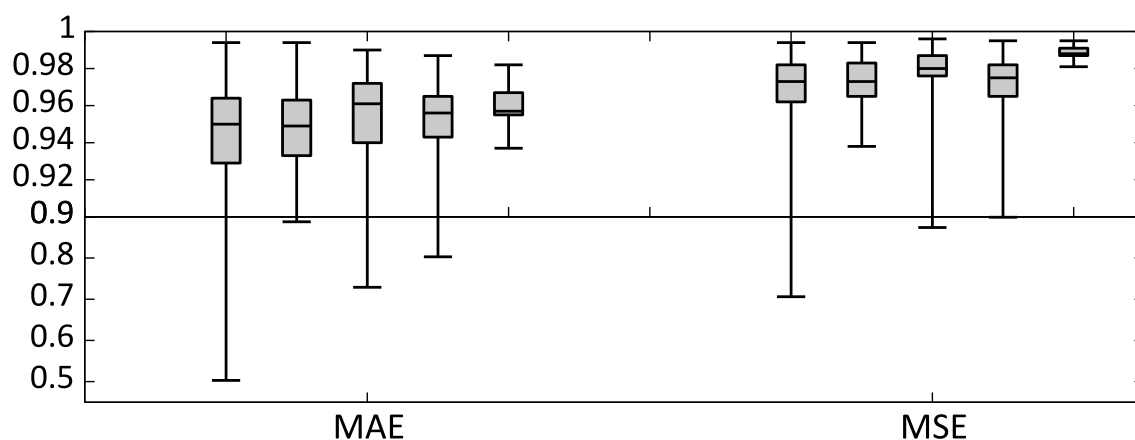


Figure 3.8: Normalized pixel-by-pixel screenshot similarity between REMAUI input and generated application on the 488 subjects, shown by group A–E from left to right. Higher values are better.

Since REMAUI currently removes all standard OS status and navigation bars from input screenshots, we do the same to the screenshots of REMAUI-generated applications. To ensure that input and generated screenshots have the same dimensions, we set the target application screen dimensions to account for subtracting the OS navigation bar.

We used the open source Photohawk³ library to measure two widely used picture similarity metrics [47]. Specifically, following are the mean absolute error (MAE) and the mean squared error (MSE) over a screenshot’s n pixels; $e_{i,j}$ is the delta of one of the three color channels RGB of a given pixel in the original vs. the corresponding pixel in the REMAUI-generated screenshot.

$$\text{MAE} = \frac{1}{3n} \sum_{i=1}^n \sum_{j=1}^3 |e_{i,j}| \quad \text{MSE} = \frac{1}{3n} \sum_{i=1}^n \sum_{j=1}^3 e_{i,j}^2$$

Figure 3.8 shows the normalized (to $[0, 1]$) similarity measures for our 488 subjects, arranged by our five groups. The results indicate that REMAUI-generated applications achieved high average pixel-by-pixel similarity with the respective inputs on both metrics.

3.6.4 RQ3: UI Hierarchy Similarity

Achieving high pixel-by-pixel similarity is not sufficient, as it is trivially achieved by a UI that consists of a single bitmap (i.e., the input screenshot). So in this section we also evaluate how closely the generated view hierarchy resembles the view hierarchy that produced REMAUI’s input screenshot.

Evaluating the quality of a given UI hierarchy is hard. First, there are often several different hierarchies of similar overall quality. Not having the application’s specification, it is not clear which alternative REMAUI should target. Second, unlike for RQ2, the input application’s UI hierarchy is often not an obvious gold standard. Many of the subjects contained redundant intermediate containers that have the same dimension as their immediate parent and do not seem to serve a clear purpose.

³<http://datascience.github.io/photohawk>

Other container hierarchies could have been refactored into fewer layers to speed up rendering.

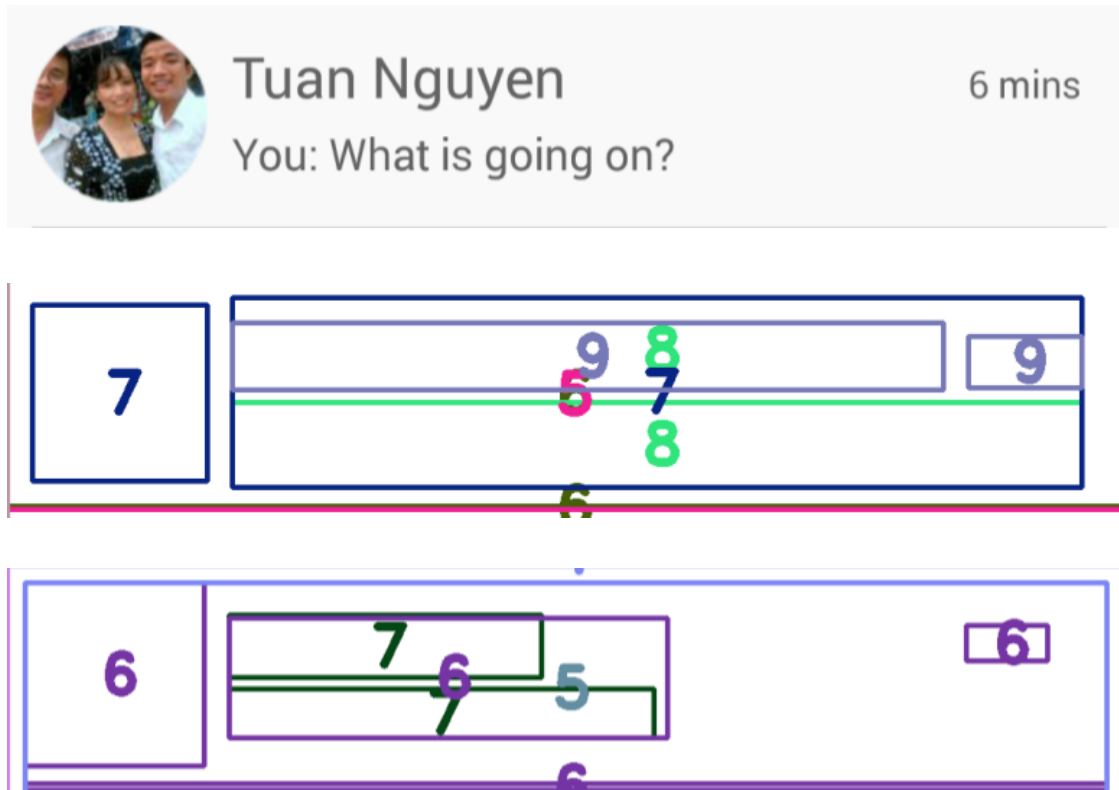


Figure 3.9: Part of a screenshot of Google Hangout (top), its UI hierarchy (middle), and the REMAUI-generated hierarchy (bottom). Each element is annotated at its center with its level in the UI hierarchy, with root=1. Each number's color matches the color of its element's boundary.

Figure 3.9 shows an example of this challenge. The original UI hierarchy and the REMAUI-generated one differ in several aspects. For example, REMAUI puts the contact's name and message into two relatively small level-7 text boxes. The original application puts the same strings into much larger level-8 text boxes. Similarly, REMAUI groups name and message into a level-6 container. The original application

groups them with the date into a level-7 container. This container is nested into a level-6 container, which is nested into a level-5 container of the same dimensions. The latter container thus seems redundant. Despite these differences, screenshots of the two hierarchies are very similar pixel-by-pixel.

In our evaluation we side-step these challenges by comparing UI hierarchies at the leaf level. While this comparison does not capture the entire hierarchy, it still captures parts of the UI’s structure. For example, the boundary of each intermediate (container) node is represented by the leaf nodes it contains.

Specifically, for this experiment we analyzed each pixel in a REMAUI-generated screenshot. If a pixel belongs to a text box in both the original and the generated application, then we consider the pixel correct. Similarly, the pixel is correct if it belongs to an image view in both the original and in the generated application. Given these criteria, we can define precision p and recall r as follows, separately for images, text, and overall, given the pixels i in an image view in the original application and in the generated application (i') as well as the pixels t in a text view in the original application and in the generated application (t').

$$\begin{aligned}
 p_i &= \frac{|i \cap i'|}{|i'|} & p_t &= \frac{|t \cap t'|}{|t'|} & r_i &= \frac{|i \cap i'|}{|i|} & r_t &= \frac{|t \cap t'|}{|t|} \\
 p &= \frac{|i \cap i'| + |t \cap t'|}{|i'| + |t'|} & r &= \frac{|i \cap i'| + |t \cap t'|}{|i| + |t|}
 \end{aligned}$$

Since in our setup this experiment required manual steps for capturing the UI hierarchies we restricted the scope of the experiment to our core group of Android subjects (group C) and the relatively small group of iOS subjects (group B). Figure 3.10 shows the experiment’s results. We found a moderate to high structural match (in terms of the leaf nodes) between original and REMAUI-generated UI hierarchies.

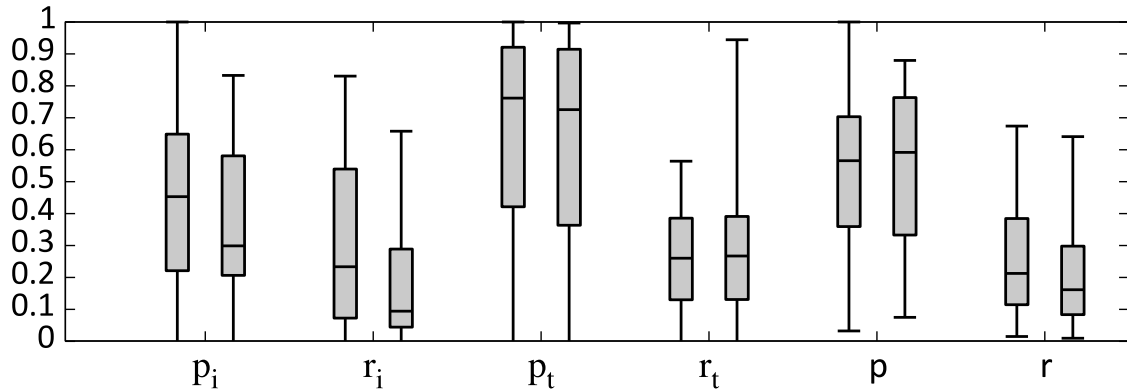


Figure 3.10: Image, text, and overall UI element precision (p) and recall (r) for groups B (left) and C (right). Higher values are better.

The low recall in Figure 3.10 does not fully capture how REMAUI reproduced text or images. On the contrary, REMAUI’s pixel-by-pixel similarity was high (Figure 3.8). We suspect a culprit of low recall was white-space. REMAUI computes tight bounding boxes, but a corresponding original text or image view may contain much additional white-space and fill its parent container (as the much larger text boxes in Figure 3.9).

To explore this issue on the example of text, we measured the Levenshtein distance (edit distance) [52] of text box strings between original and generated applications. For the 2.4k string pairs of group C, on average, an original text was 18.8 characters, a generated text 14.6, and the edit distance 4.7. So on average it took only 4.7 single-character additions, removals, or substitutions to convert a string in the generated application back to the corresponding string in the original application. For the 2.9k group B string pairs, on average, an original text was 14.6 characters, a generated text 15.0, and the edit distance was only 2.9. These results indicate a higher text recall than the pixel-based recall of Figure 3.10.

The following two trends emerged on manual inspection. First, precision suffered if a subject contains a bitmap that contained both text and non-text imagery.

This is not surprising, as for OCR it is hard to distinguish if a given text is plain text or belongs to a bitmap of text and other elements. REMAUI typically decomposed such bitmaps into text and image views. The resulting UI hierarchy should be relatively easy to fix manually. A developer would just replace a generated container (containing both text and images) with a single bitmap. Overall these incorrectly detected views were small. In Figure 3.10, their average area was less than 0.25% of the input screen area.

The second observation is that low image recall occurred when images overlapped that were of similar color or where the top image is somewhat transparent. These scenarios are challenging for edge detection. Similarly, text recall was low if the text color was similar to the background color. On the flip-side, with high contrast we observed high recall.

3.7 Related Work

The gap between early prototyping and formal layout definition also exists in the related domain of web site development. A study of 11 designers at 5 companies showed that all designers started with sketching the layout, hierarchy, and flow of web pages with pencil on paper and in graphical design tools such as Photoshop [53].

A similar design process has been reported for desktop applications. At Apple, user interface sketches were first created with a fat marker (to prevent premature focus on details) and later scanned [54]. Separate studies of hundreds of professionals involved in UI design in various companies indicated heavy use of paper-based sketches [55, 56]. One of the reasons was that sketching on paper is familiar due to designers' graphic design background.

Despite much progress in tools for creating user interfaces that combine the unique talents of graphic designers and programmers [57, 58], much conceptual user

interface design work is still being done by graphic designers with pencil on paper and digitally, e.g., in Photoshop. Previous work has produced fundamentally different approaches to inferring user interface code, as it was based on different assumptions. Following are the main changed assumptions for mobile application UI development and reverse engineering that motivate our work. (1) First, many UX designers and graphic artists do not construct their conceptual drawings using a predefined visual language we could parse [59, 60, 61, 62]. (2) Second, while this was largely true for desktop development, mobile application screens are not composed of the platform’s standard UI framework widgets [63, 64]. (3) Finally, we cannot apply runtime inspection [65, 66] as REMAUI runs early in the development cycle.

Specifically, the closest related work is MobiDev [62], which recognizes instances of a predefined visual language of standard UI elements. For example, a crossed-out box is recognized as a text box. But unlike REMAUI, MobiDev does not integrate well with a professional mobile application development process. It would require UX designers and graphic artists to change the style of their paper and pencil prototypes, for example, to replace real text with crossed-out boxes. Such changes may reduce the utility of the prototypes for other tasks such as eliciting feedback from project stakeholders. In a traditional reverse engineering setting, MobiDev cannot convert screenshots into UI code.

SILK and similar systems bridge the gap between pen-based GUI sketching and programming of desktop-based GUIs [59, 60, 61]. Designers use a mouse or stylus to sketch directly in the tool, which recognizes certain stroke gestures as UI elements. But these tools do not integrate well with current professional development processes as they do not work on paper-on-pencil scans or screenshots. These tools also do not recognize handwritten text or arbitrary shapes.

UI reverse engineering techniques such as Prefab [63] depend on a predefined model of UI components. The work assumes that the pixels that make up a particular widget are typically identical across applications. However, this is not true for a mobile application UI. Mobile applications often have their own unique, non-standard identity, style, and theme. For Prefab to work, all possible widget styles and themes of millions of current and future mobile applications would need to be modeled.

PAX [65] heavily relies on the system accessibility API at program runtime. At runtime PAX queries the accessibility API to determine the location of text boxes. The accessibility API also gives PAX the text contents of the text box. PAX then applies computer vision techniques to determine the location of words in the text. If a view does not provide accessibility, PAX falls back to a basic template matching approach. PAX thus cannot reverse engineer the UI structure of mobile applications from their screenshots or application design images alone.

Recent work applies the ideas of SILK and DENIM to mobile applications [67], allowing the user to take a picture of a paper-and-pencil prototype. The tool allows the user to place arbitrary rectangles on the scanned image and connect them with interaction events. This idea is also implemented by commercial applications such as Pop for iOS. As SILK and DENIM, this approach is orthogonal to REMAUI.

3.8 Threats to validity

3.8.1 Judging the quality of our tool output on concept drawings is subjective

The conceptual drawings used in this paper have not yet been implemented in an application, so evaluating the correctness of its generated application is subjective. Moreover, when UX designers work on the conceptual drawing, he/she does not think about the low level source code implementation. From one version the conceptual

drawing, a programmer can have several ways to implement it. Thus, the generated code of the implementation from the conceptual drawings may be different from its actual implementation. However, in any case, the generated source code always has good value for programmers to get started.

3.8.2 The metrics we picked for comparing tool output to screenshots may not capture the “real” quality

With image similarity method (using MAE and MSE metrics), in the worst case, REMAUI reverse engineering screenshot as one large image view. This will result in the highest value in Figure 3.8. However, our source code structure can still be verified using hierarchy comparison method. And MAE and MSE metrics still have value to support evaluate REMAUI in visual way.

3.8.3 We picked top-100 iOS apps. These may be biased towards simpler UI

The top-100 iOS apps is highest ranking apps in the iOS App Store. This app set may not capture the rest of mobile application. Moreover, these perceived quality apps may have simple UI in comparison with less perceived quality app [7]. We also evaluate our tools on Google and Apple apps which are well-designed apps from top two largest mobile software company.

3.9 Conclusions and Future Work

When developing the UI code of a mobile application, a big gap exists between graphic artists’ conceptual drawings and working UI code. Programmers typically bridge this gap manually, by reimplementing the conceptual drawings in code, which is cumbersome and expensive. To bridge this gap, we introduced the first technique to automatically Reverse Engineer Mobile Application User Interfaces (REMAUI). On

a given input bitmap REMAUI identifies UI elements via computer vision and OCR techniques. In our experiments on 488 screenshots of over 100 popular third-party applications, REMAUI-generated UIs were similar to the originals, both pixel-by-pixel and in terms of their runtime UI hierarchies.

We plan to (1) generalize the export step to additional platforms such as iOS and cross-platform JavaScript-based frameworks. (2) REMAUI currently converts each input screen to a separate application. We plan to provide a graphical notation to allow users to connect several input screens drawings, which REMAUI could use to generate a single application with various screens and corresponding screen transitions. (3) We plan to integrate REMAUI with tools that generate mobile application functionality either via keyword-based code search [68] or from high-level models [69, 70, 71] or DSLs [72, 73]. (4) We plan to index a screenshot corpus by running REMAUI on it and storing REMAUI’s intermediate results. Exposing this index via a query interface would allow a user to search for screenshots by their structure and features.

The REMAUI prototype implementation for Android used in the evaluation is freely available on the REMAUI web site: <http://cseweb.uta.edu/~tuan/REMAUI/>

CHAPTER 4

REGLADOM: REVERSE ENGINEERING LEGACY APPLICATIONS INTO HIGH-LEVEL DOMAIN MODELS

4.1 Overview

Automating this legacy software system reverse engineering task poses two key challenges, which existing tools and techniques do not fully address. Challenge (1) is to automatically point the reverse engineer to those parts of the application that contain the application’s core business logic. I.e., in an object-oriented application some objects are more interesting than others. There is typically a set of classes that represents key domain concepts. These are the classes a reverse engineer wants to focus on early, to get a quick overview of the core business logic of the analyzed application. Besides those interesting key domain concept classes, a large program has many more classes that are less interesting. We call these less interesting classes, “plumbing” classes, as their typical task is to connect core domain classes to other parts of the system.

Distinguishing between core and plumbing classes automatically is hard in general. To make that distinction, a reverse engineering tool has to understand what it means to be part of the business logic. However, each analyzed program supports a different business logic and implements that logic in its own way, with its own naming conventions, domain language, acronyms, etc. We found that current tools and techniques are limited in that they either ignore the distinction between core and plumbing classes [74], [75] or manually select core classes then find their detail information and relations [24]. In this paper, we step back and try to understand

the general principles that domain concept classes share at the implementation level. I.e., how domain concept classes express their behaviors and their relations with each other in source code, and where parts of the application source code contain more precise domain concept information than others. In our evaluation, our technique compared with the most closest related previous approaches, our technique is able to identify the core domain model classes much more precisely. I.e., on 19 open-source applications, we have measured 1.92 times increase in precision and 3.58 times increase in recall.

Challenge (2) broadly originates from the conceptual differences between the relational style of the high-level design models and the procedural and object-oriented style of the low-level software code. In software code, data is typically maintained in a network model. Data is stored in a distributed fashion in many individual objects, which keep their data private. Many associations between objects are subtle and only manifest when methods operate on this object graph. In contrast, in the high-level relational model, all data is stored in a few large relations, each of which can contain data of millions of individual objects. The challenge is to analyze the low-level software in a deep enough fashion to extract those relations that are important but may be encoded in the code in non-obvious ways.

Evaluation of reverse engineering approaches such as RengLaDom is a well-known challenge [18]. The key problem in is how to evaluate the correctness of RengLaDom approach on reverse engineering the domain model since many applications is lack of verified and an accurate domain model. To address this evaluation challenge, we describe a new approach to evaluating reverse engineering tools. Our approach takes advantage of existing domain models that are embedded into the application code. By using the same technique, we also show that RengLaDom is successfully used to reverse engineer wide variety of non-ORM applications.

In summary, this paper makes the following contributions.

- A technique for pinpointing domain model classes.
- An implementation of these techniques in an open-source *RengLaDom* (*Reverse engineering Legacy applications into high-level Domain models*) tool for Java, which is available at <http://cseweb.uta.edu/~tuan/rengladom/>
- A new technique for evaluating reverse engineering tools.
- An evaluation that shows that RengLaDom infers domain model classes that are of higher quality than those inferred by the most closest related reverse engineering techniques.

4.2 Overview and Examples

At a high level, we identify domain model classes and infer subtle domain relations by performing several passes over two graphs, the method call graph and the graph that maps each method to the set of classes used by that method. We found that these two graphs provide us with the right trade-off between deep semantic information and useful abstraction, which enables a light-weight yet semantically meaningful analysis.

4.2.1 Which are the domain model classes?

Domain model classes are used to store important business relations. Business relations are implemented as a network of objects [76].

When developing an object-oriented program, developers are encouraged to first identify the key items or concepts of the program's target domain [76, 77]. For example, a pizza delivery application will have to deal with pizza orders and toppings, which yields the first two concepts, pizza orders and toppings. Developers then typically map each domain concept to an implementation class, such as a Java class or

interface, for example, yielding the Java classes `PizzaOrder` and `Topping`. The code thus represents a real-world topping with a `Topping` class instance.

In addition to items or concepts, developers also identify how these items are related to each other. In other words, developers describe relations on the concepts. During development this often yields an informal UML class diagram or a conceptually very similar artifact from the database community such as an entity-relationship diagram. In our example, the developer may identify that a pizza order includes N toppings. Each such relation is then often mapped to code as instance fields. In our example, the `PizzaOrder` class may have an instance field that refers to a collection of `Topping` instances.

In addition to domain model classes and their relations, to make the application work, developers also need a lot of classes other than domain model classes. We call such classes *plumbing classes*. Examples of plumbing classes are controller classes when implementing the Model View Controller pattern, configuration classes, utility classes, and classes that represent external resources. Such classes are required in an application to connect domain model classes with each other and with external libraries. Examples of plumbing classes in the pizza application include the `DbDAO` and `ServiceException` which handles database utility works and errors or exceptions during program execution classes.

On the surface, it is hard to distinguish plumbing classes from domain model classes. Standard metrics such as class size, complexity measures, number and type of fields and methods, relative position in the subtype-hierarchy, and naming can typically not reliably distinguish a domain model class from a plumbing class. For example, in the Pizza Application, the name of the `SysTime` class may suggest that it is a plumbing class. On the other hand, the name of the `StudentService` class may suggest that `StudentService` is a domain model class. However, according to the

Pizza Application documentation, in both cases the opposite is the case and the other metrics are not reliable either.

Our hypothesis is that a domain model class can be identified and thereby distinguished from a plumbing class with high probability by observing the following three characteristics: (a) in how many execution sequences the application uses a class together with other classes, (b) how the relations of the class with other classes are stored, and (c) how these relations are read and updated by other classes. Our evaluation shows that these three characteristics are indeed powerful predictors for identifying domain model classes. Before we make our hypothesis more precise and test it, we first provide our intuition that motivates these three characteristics.

4.2.1.1 Domain model class characteristic 1

In how many program execution sequences does the application use a class together with other classes? This characteristic is a good proxy for being part of the domain model, if we assume that the domain model classes are at the core of an application. Then the application tends to use a domain model class in several execution sequences, satisfying various business needs. Contrast that with a plumbing class that may be very focused. For example, a use case controller class has the sole task of controlling a single use case. Such a plumbing class will therefore not participate in a large number of execution sequences as a domain model class does.

It is an interesting question what it means for an application to use a class or type in an execution sequence. When implementing a program, a programmer can often choose between many different ways of implementing a particular programming task. We assume that a programmer tends to choose an option that documents the important concepts involved in a task. Often the important concepts are domain model concepts. For example, a programmer could create a local variable to store a

domain model class instance such as in the first statement of the method of Listing 4.1. Having a `Topping` as an explicit local variable documents in the source code that `Topping` is an important concept in this method. On the other hand, if the thereby created `Topping` instance is only used once in this method, the constructor call could also be written inline in that subsequent expression, passing the `Topping` to another method. Creating the `Topping` inline would deemphasize the use of `Topping` in this method. We assume that programmers tend to emphasize the use of domain model classes in their code and therefore explicitly create local variables of a domain model class type.

Besides local variables, a program uses data and types in various styles. The two styles we also consider are data passed as a method parameter and data stored in instance fields. In summary, we assume that the more often a program uses a type as the type of a local variable explicitly declared in the source code, the type of a method parameter, or the type of an instance field, the more likely it is that the type represents a domain concept. When we say that an execution sequence or method uses a class `X` we mean one of these three options—an instance of `X` is used either in an explicitly named local variable, method parameter, or instance field.

Since a domain model class is at the core of an application and has several relations with other domain model classes, it is likely that an execution sequence that uses a given domain model class also uses other classes—especially, other domain model classes. In contrast, a utility class may never be instantiated or, if instantiated, may not be used in the same execution sequence with another class.

For example, in the `Pizza` application, class `Topping` is used in `PizzaOrderDAO.findToppingByName(String)` to find all topping by the given name. In contrast the plumbing class `HibernateUtil` is only used as static references.

```

public Topping findToppingByName(String toppingName)
{
    List<Topping> tops = dbDAO.getSession().createQuery("from Topping t where t.
        toppingName = '" + toppingName + "'").list();
    return tops.get(0);
}

```

Listing 4.1: The findToppingByName method. Topping is a domain model class and is stated prominently as the (parameter of a) type of an (explicitly named) local variable.

4.2.1.2 Domain model class characteristic 2

How does the application store the relations a class has with other classes? This characteristic is a good proxy for being part of the domain model as object-oriented design guidelines prescribe storing each such relation in instance fields. Instance fields are not the only option for storing relations between objects at runtime, as, for example, another class could store such a relation in a hash table. However, programmers seem to follow the design guidelines and store relations between domain model classes in instance fields.

For example, class PizzaOrder stores a list of Topping instances. In contrast, the plumbing class HibernateUtil does not store any relations in instance fields.

4.2.1.3 Domain model class characteristic 3

How are the relations between class instances read and updated by other classes? This characteristic is a good proxy for being part of the domain model as this domain model contains information that is very valuable to the application. Other application

classes therefore likely need to access this domain model information to read and update its current values.

For example, class `PizzaOrder` has methods to read and write the values of its instance fields. In contrast, the value of the plumbing class `HibernateUtil` cannot be updated by other application classes during execution.

With this hypothesis and its three component characteristics set out, we now translate the component characteristics from their high-level description into a lower-level operational model that can be checked for existing applications.

How often a class is used with other classes we can observe by examining the application's method call chains. According to our hypothesis, if a class `X` appears in many call chains together with other classes, then the class is likely important for the application and a domain model class. How the relations of the class with other classes are stored and how relations between class instances are read and updated by other classes?

If application class `A` is to be a domain model class, then there has to be another class `B` in the system, such that `A` and `B` are used¹ together by a single method or a single method call chain. We use this test to approximate the set of domain model classes, by analyzing how the application uses classes throughout the method call graph. In other words, if an application class `C` is never used with any other application class `X`, either in the same method or in the same call chain, then `C` is likely not a domain model class.

This domain model class test so far is binary, it separates all application classes into two groups, domain model classes and plumbing classes. We can refine this test into a ranking scheme that computes the number of times a given class is used

¹variables whose static types, either `A` or `B`, are used in the method in three cases: parameter declaration, field access, and local variable declaration

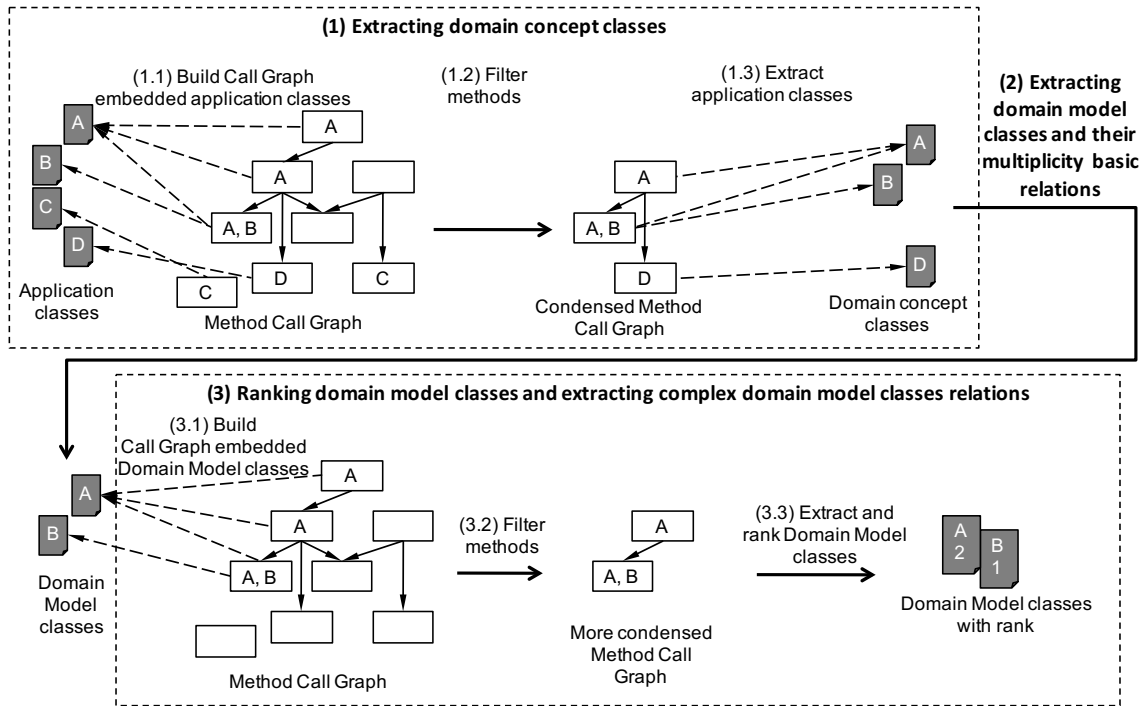


Figure 4.1: RengLaDom is a three-step approach to automatic domain model reverse engineering. RengLaDom starts with identifying domain model classes. On those classes, it then identifies both basic and complex relations. (1) Extracting domain concept classes (2) Extracting domain model classes and their multiplicity basic relations (3) Ranking domain model classes classes relations

together with other application classes. One domain model class are more important than others if it has more interactions with other domain model classes.

In order to reverse engineering domain model classes and their relations, we build RengLaDom as a filter mechanism with three-step approach. See Figure 4.1. (1) RengLaDom try to find classes which describe important concepts of the application among all classes in the application. (2) Then it will extract only classes are more intelligibly domain model classes, and also extract it attributes and multiplicity relations. (3) Finally, it ranks the importance of domain model classes.

4.2.2 Motivating Example: Pizza Shop

Consider the following example Java application, Pizza Shop [78]. Pizza Shop lets students order pizza with different sizes and toppings for delivery to specific dorm rooms and maintains its state in a relational database. We choose this application for its following two features. First, Pizza Shop has a built-in mapping from objects to relations, in the form of a Hibernate mapping [79]. Second, Pizza Shop is better documented than typical third-party Java applications, as it is part of a SIGMOD tutorial² [78], which describes the Pizza Shop requirements and design alternatives. This allows us to compare the given object-relational mapping and the domain models reverse-engineered by various research tools with the domain model envisioned by the original Pizza Shop creators.

Figure 4.2 shows Pizza Shop’s database schema as an entity-relationship (ER) diagram [80]. Pizza Shop maintains one N-N relation between `PizzaOrder` and `Topping` as well as one N-1 relation between `PizzaOrder` and `PizzaSize`. Like many other applications, Pizza Shop represents a subset of its database state in the dynamic state of the program, in individual program objects. Listing 4.2 shows that for each entity of the ER diagram, the Pizza Shop code has a corresponding Java class. The ER diagram relations are represented in the objects as fields. For example, the N-N relation between `PizzaOrder` and `Topping` is encoded both in the `PizzaOrder` class (with a set-of-`Topping` field) and in the `Topping` class (with a set-of-`PizzaOrder` field).

```
public class PizzaOrder { // ..  
  
    int id;  
  
    PizzaSize pizzaSize;  
  
    int day;  
  
    Set<Topping> toppings = new HashSet<Topping>(0);
```

²<http://www.cs.umb.edu/~eoneil/orm/>

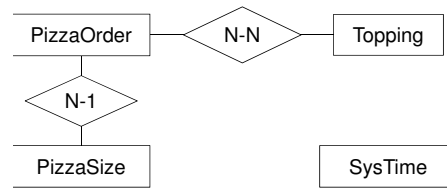


Figure 4.2: Entity-relationship diagram of Pizza Shop’s relational database. There are four entities, PizzaOrder, Topping, PizzaSize, and SysTime. There is an N-N relationship between PizzaOrder and Topping and an N-1 relationship between PizzaOrder and PizzaSize. SysTime does not have direct relationship with PizzaOrder, Topping, or PizzaSize but it is used to track time in days which links with *day* property in PizzaOrder.

```

}
public class Topping { // ..
    int id;
    Set pizzaOrders = new HashSet(0);
}
public class PizzaSize { // ..
    int id;
    Set pizzaOrders = new HashSet(0);
}
public class SysTime { // ..
    int id;
}
  
```

Listing 4.2: Pizza Shop source code, abbreviated for space reasons (in the code each field is private and each class has additional members).

In total, Pizza Shop contains 18 Java classes. From those 18, according to the Pizza Shop documentation, the domain model consists of four classes, PizzaOrder, PizzaSize, Topping, and SysTime. Existing reverse engineering tools do not focus on deriving from all application classes the subset of likely domain model classes. I.e.,

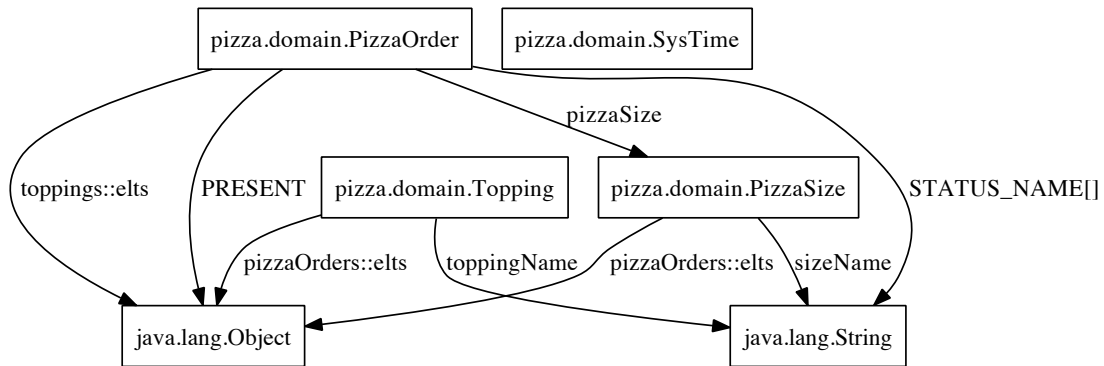


Figure 4.3: Output of Womble tool in Pizza Shop project

neither of the related tools Womble [24] and class2uml [25] select from the set of all classes those they deem likely to be the core domain model classes. In contrast, RengLaDom performs several passes over the Pizza Shop source code and presents the user a ranked list of the most likely domain model classes³ In this example, RengLaDom reliably ranks highest the four classes that are considered the domain model classes in the Pizza Shop documentation.

4.3 Background

4.3.1 Analyzing only method bodies is enough to know about domain concept

4.3.2 Methods using classes

We consider three kinds of variable whose static type is a domain model class. These variables are used in the method in three cases: (1) parameter declaration, (2) field access, and (3) local variable declaration. See (Listing 4.3).

```
class C {
```

³In doing that, RengLaDom does not take into account the Hibernate mapping. I.e., RengLaDom produces the same output when we remove all Hibernate mapping information from the analyzed application.

Hibernate Mapping	Womble	RengLaDom
<i>PizzaSize</i>		
1:n PizzaOrder	1:n Object	1:n PizzaOrder
<i>PizzaOrder</i>		
n:1 PizzaSize	1:1 PizzaSize	n:1 PizzaSize
n:m Topping	1:n Object	n:m Topping
<i>SysTime</i>		
		1:n PizzaOrder
<i>Topping</i>		
n:m PizzaOrder	1:n Object	n:m PizzaOrder

Table 4.1: Comparison Domain Model Class Relation defined by Hibernate Configuration and the ones reversed engineered by RengLaDom and Womble. With RengLaDom, we run it on both the original Pizza Shop application and on a version from which we stripped the Hibernate persistence.

```

A fieldA;

//(1) Parameter declaration.
m2(A paraA){
}

m1(){
    //(2) Field access.
    fieldA.doSomething();
    //(3) local variable declaration
    A localA;
}
}

```

Listing 4.3: Examples of variables whose static types are a domain model class

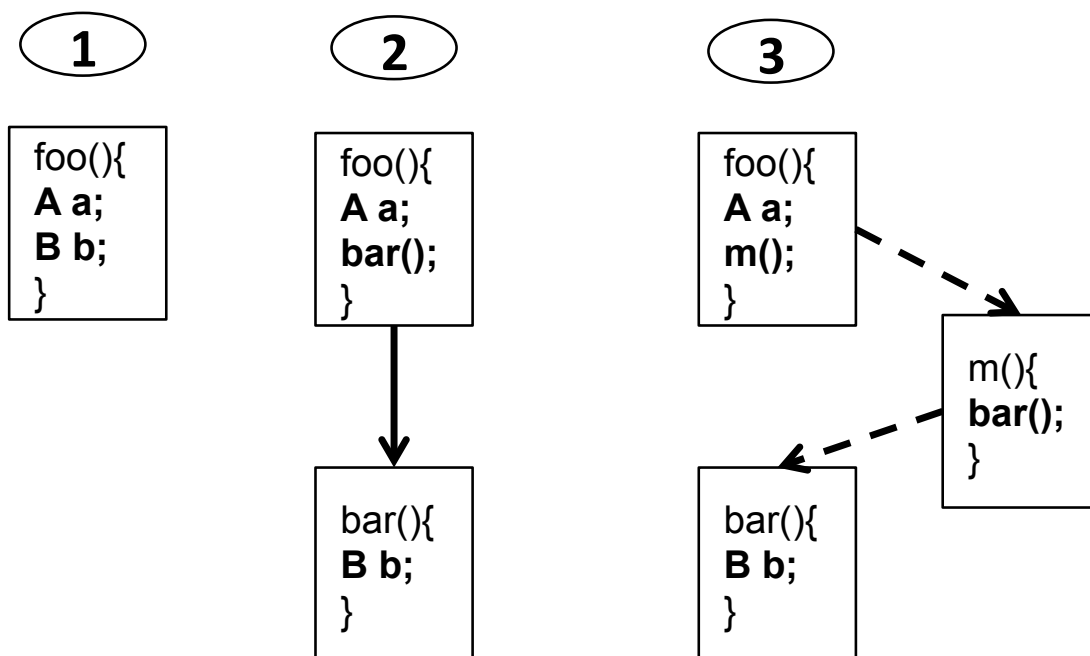


Figure 4.4: All cases of a method access to objects and their fields to start navigating a business relation: (1) method `foo` directly accesses object of domain model class `A` and `B`, (2) method `foo` access object of domain model class `A` and directly calls a method `bar` that accesses a `B` domain object, (3) method `foo` access object of domain model class `A` and transitively calls a method `bar` that accesses a `B` domain object.

4.4 Implementation

4.4.1 Overview

The following explains three-step approach to automatic domain model reverse engineering is used by RengLaDom.

(1) Extracting being-used-together classes, (Step 1 of Figure 4.1): With a given set of all classes in the application, application classes, as the input, RengLaDom can distinguish being-used-together classes, which describe important domain concept of the application from application classes. We notice that the being-used-together classes not only describe important concepts of the application but also strongly connected through the method call graph. I.e., for each pair of being-used-together

classes A, B, there has to be a pair of methods m1, m2, such that m1 uses A, m2 uses B, and m1 calls m2 or m1 is m2. This "calls" in the above includes "transitive calls", for example, if m1 calls m3, m3 calls m2, we still consider that m1 calls m2. Figure 4.4 shows all three possible examples of the relation between method call graph and being-used-together classes.

At this step, RengLaDom separates all application classes into two groups, being-used-together classes and plumbing classes. At step 1.1, RengLaDom creates method call graph of all methods in the application and add to each method vertex the set of application classes used by that method. At step 1.2, basing on the number application classes in method vertexes, RengLaDom removes method call chains that do not use more than one classes, from method call graph. At step 1.3, RengLaDom extracts application classes remain on the condensed method call graph. With our hypothesis, these remained application classes are being-used-together classes.

(2) Extracting domain model classes and their multiplicity relations (Step 2 of Figure 4.1): With a given set of being-used-together classes as the input, RengLaDom keeps only classes contain at least one field, which has both getter and setter. This means that these following things must be satisfied: classes need to contain (1) at least a field "f", (2) a method writes "f" and (3) a method read "f". They are called domain model classes. RengLaDom also analyzes fields of domain model classes to reverse engineering multiplicity relations.

(3) Ranking domain model classes (Step 3 of Figure 4.1): this step is similar to step 1. At step 3.1, RengLaDom reuses the condensed method call graph from output of step 1.2. Then it adds to each method vertex the set of domain classes which are used by that method. These domain model classes are from output of step 2. At step 3.2, basing the number of domain model classes in method vertexes, RengLaDom also removes method call chains that do not use more than one classes. The output

method call graph in 3.2 is more condensed than the one in step 1.2. At step 3.3, RengLaDom counts the present of each domain model class among all method bodies in the condensed method call graph. It uses this count as the rank domain model classes

4.4.2 Method Call Graph

The method call graph in step 1.1 contains all explicit method calls (invokevirtual, invokestatic, invokespecial, invokeinterface) within the analyzed method bodies. Calls occurring in native code are not captured in that graph. An interesting special case of such native calls is calls the analyzed code may make using Java reflection. Example of invoking method by using Java reflection which is described in Listing 4.4 is example of invoking method by using Java reflection. `java.lang.refelect.Method.invoke` calls native code and therefore such call is not shown in the graph, even though it originated in user code.

```
Class cls = Class.forName("pizza.domain.PizzaOrder");  
2 // create reflection method object  
Method method = cls.getMethod("getId", new Class[0]);  
4 // invoke getId method  
Object returnObject = method.invoke(cls.newInstance(), new Object[0]);
```

Listing 4.4: Small Java reflection example of invoking `getId` method of `pizza.domain.PizzaOrder` class

Figure 4.5 is an example of method call graph in which each vertex is a method which has method name on top, and follows by variables that have a static type that is a domain model class (format: `DomainModelClass variableName: relationType`).

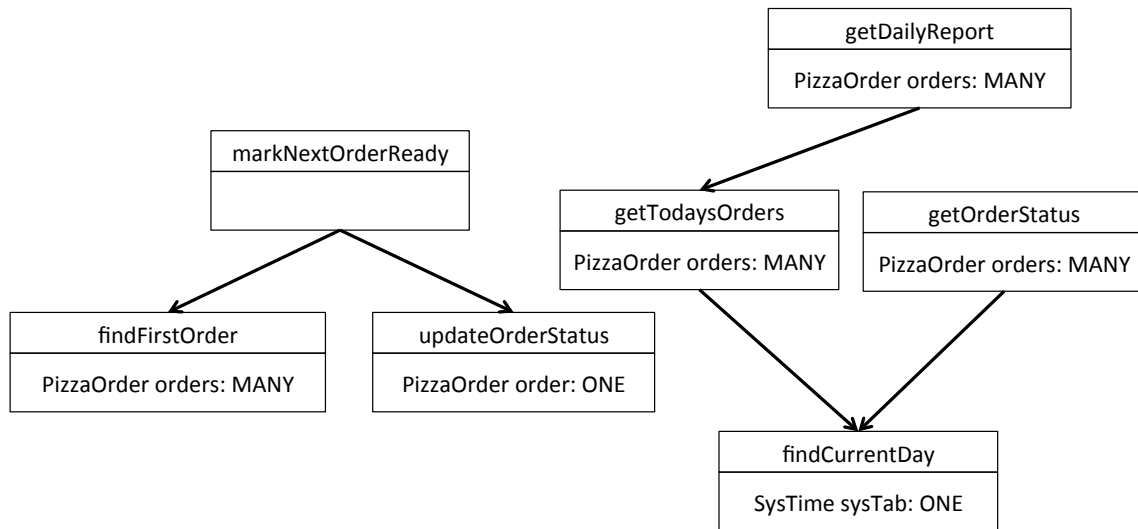


Figure 4.5: Example of method call graph

There are 2 two kind of relation type of a variable: single relation type (ONE) and container relation type (MANY).

4.4.3 Filtering Methods

The aim of these filters is to remove unimportant methods from method call graph. The method filter is bases on references of input classes to rank method vertexes in method call graph. The more reference of input classes in a method and its successors the higher of the rank of method vertexes. The method filter will base on these rank to transfer and condense method call graph by using different elimination criteria: removing (1) solitary vertexes, (2) empty root subgraphs, (3) empty leaf subgraphs, and (4) root delegated vertexes. In this step, the method call graph is transferred to several filters in sequence.

Solitary method filter The filter bases on two criteria: (1) methods are solitary methods, and (2) methods do not contain significant business logic to removes method vertexes in method call graph. Solitary methods methods is methods that

have callDegree 0 ($callInDegree = callOutDegree = 0$). Unused methods, which are not called by any other method, are solitary methods. Unused methods have callInDegree equals 0. However, in some cases, callDegree of method is 0, but it may be executed by using reflexion. In this situation, the second criteria (2), containing significant business logic, is used. This means that the callInDegree 0 methods should not contains statements that refer to domain model classes ($|C_{Dm}| = 0$). If first two conditions are satisfied, the method may still have successors that contain significant business logic. One more condition is added, the callOutDegree of the method must equals 0. The unused methods with callOutDegree 0 becomes solitary methods. In special cases, 2 variables have the same static type that are a domain model class can appear in a method. We separates them from this section, so C_{Dm} of solitary methods must be less than or equal to 1. We denote the set of these method as set M_a and two criteria of the filter can be defined as the following formular.

$$M_a = \{m \in G_M(V) | deg_{G_M}^+(m) = 0 \quad (4.1)$$

$$\wedge deg_{G_M}^-(m) = 0 \wedge |C_{Dm}| \leq 1\} \quad (4.2)$$

$$G_M(V) = G_M(V) \setminus M_a \quad (4.3)$$

Bottom-up empty subgraph filter The filter bases on two criteria: (1) methods that does not contain any domain model class, and (2) their successors do not contain any domain model class to removes bottom subgraph in method call graph. RengLaDom analyzes bottom-up method call graph from leaf vertexes to their parents, and removes vertexes that do not contain any domain model classes. If a method that removed in this situation, its largest sub graph, which has it as a root

vertex, does not contain any significant business logic. We denote the set of these method as M_b .

$$M_b = \{m \in G_M(V) \mid |C_{Dm}| = 0 \quad (4.4)$$

$$\wedge (\forall n \in M_G(m, S), |C_{Dn}| = 0)\} \quad (4.5)$$

$$G_M(V) = G_M(V) \setminus M_b \quad (4.6)$$

Top-down empty subgraph filter The filter also bases on two criteria: (1) methods are not called by any other methods, and (2) do not contains any domain model classes to removes top subgraph in method call graph. This filter is different from bottom-up empty subgraph filter, the aim of this top-down filter is to remove methods which are in top of method call graph and do not contain any significant business logic. The filter removed methods are not called by any other methods ($\text{callInDegree} = 0$) and do not contains any domain model classes. We denote the set of these method as M_c .

$$M_c = \{m \in G_M(V) \mid \text{deg}_{G_M}^-(m) = 0 \quad (4.7)$$

$$\wedge |C_{Dm}| == 0\} \quad (4.8)$$

$$G_M(V) = G_M(V) \setminus M_c \quad (4.9)$$

Delegation method filter The filter bases on three criteria: (1) methods are not called by any other methods, (2) has only one direct successor, and (3) its

relations of domain model classes are equal to its only direct successor's to removes methods in method call graph. The filter removed methods are not called by any other methods ($callInDegree = 0$) and they are delegated tasks from its successor. The filter keep their successors, the actual method done tasks. We denote the set of these method as M_d .

$$M_d = \{m \in G_M(V) | m \in G_M deg_{G_M}^{-}(m) = 0 \quad (4.10)$$

$$\wedge |G_M(m, DS)| = 1 \wedge (\forall n \in M_G(m, DS), \quad (4.11)$$

$$|C_{RDm}| = |C_{RDn}|) \} \quad (4.12)$$

$$G_M(V) = G_M(V) \setminus M_d \quad (4.13)$$

domain model class variable pair filter The filter bases on one criterion: an domain method must be a method contain at least two variables of domain model classes. These variables can present either in the method or in its successors and they belong to set $|C_{EVDm}|$. We denote the set of these method as M_e .

$$M_e = \{m \in G_M(V) | |C_{EVDm}| \geq 2 \} \quad (4.14)$$

$$M_{DA} = M_e \quad (4.15)$$

public method filter This filter bases on one criterion: an domain method must be a public method . We assign $M_{DA} = M_e$.

$$M_{DA} = \{m \in G_M(V) | mispublicmethods \} \quad (4.16)$$

4.5 Evaluation Methodology

Different reverse engineering tools will address these challenges in different ways. In order to evaluate our and other approaches, we propose to compare the results of such tools with existing relational models. Recently, several object-oriented applications have added such a relational model. It came in the form of a binding to a data persistence system, often a relational database. Several frameworks such as Hibernate [79] have emerged that support such object-relational mapping (ORM). Taking advantage of such an existing mapping allows us to compare an inferred relational domain model with a domain model that was considered valuable by the original forward engineers of the application.

“A simple Domain model looks very much like the database design with mostly one domain object for each database table” [81, p 117]

4.5.1 Voting

Run several tools in parallel and declare majority of results to be the true result. This has been done for evaluating 11 Java-to-UML reverse engineering tools [82].

4.5.2 Compare Analysis Results with Existing Object-Relational Mapping (ORM)

Need for rigorous evaluation has been much discussed [83].

Todo: Explain that ORM mappings have recently become popular in object-oriented development. Popular frameworks include Hibernate. That means that several applications come with ORM mappings. ORM mappings can be in form of class annotations or Xml files. We compare our analysis results with the information in these existing Xml files.

Todo: Talk about similarity between database relations and relational domain models such as UML class diagrams.

4.6 Evaluation

Todo: Discuss how our empirical evaluation relates to overview paper [84].

Todo: Classify RengLaDom using a third-party classification system such as [85], which may enable a qualitative comparison with related papers and tools.

4.6.1 Subjects: A surprising result on non-database application

jMusic⁴ is a compositional and audio processing library in Java to support composers and software developer. RengLaDom is done a great job to correctly select domain model classes which verified by author documents.

“Let’s turn our attention to the `jm.music.data` package as this is the only one that you really need to know.”⁵

“The core of the jMusic data structure are the five classes in the `jm.music.data` package contains four Java source code file:” [86]

However, The paper then lists five classes (Note, Phrase, CPhrase, Part, and Score). The entire JMusic project contains some 200 classes. Upon careful review of these five classes, we believe that the most important classes are Note, Phrase, Part, and Score. Our tool identifies exactly those classes.

SweetHome3D⁶ is a free interior design application that helps you draw the plan of your house, arrange furniture on it and visit the results in 3D.

We try to run RengLaDom on SweetHome3D project. This is a well documented project which also include a domain model document in it Plug-in developer’s guide⁷. In SweetHome3D project, there is a domain model package which include 47 classes. However, the author’s the domain model document only mentions 27 classes. We

⁴<http://explodingart.com/jmusic/>

⁵<http://explodingart.com/jmusic/jmtutorial/x41.html>

⁶<http://www.sweethome3d.com>

⁷<http://www.sweethome3d.com/pluginDeveloperGuide.jsp>

argue that 27 classes are the most important classes in the project. After we run RengLeDom on top of 1366 classes in the entire SweetHome3D project. RengLeDom infers that there are 43 domain model classes and also ranks them by their importance. The significant outcome of the result is that the top 10 classes ranked by RegleDom is all included into the 27 classes identified by the SuiteHome3D’s author.

4.6.2 Subjects: Applications That Have An Object-Relational Mapping

The pizza project [78] is a great benchmark project because we not only have the source code but also the mapping to a relational database schema. Since database schemas are usually carefully designed, we assume that the database schema represents the right (or correct or ideal) domain model. Hence the architect’s true domain model associates `PizzaOrder` with `SysTime`. Having this “perfect” domain model as a benchmark, we can now compare different reverse engineering tools that only analyze source code, such as Womble and RengLaDom. Womble can only infer a relation between `PizzaOrder` and its `int` field called `day`. RengLaDom analyzes how values are inferred in methods and can therefore infer that the value assigned to this `int` `day` field was in fact inferred from `SysTime`.

`pizza_wo` is a version of the Pizza Project in which we have removed all dependencies on Hibernate.

`YouEat`⁸ is ..

TODO: Reference Listing 4.5 from text.

```
1 create table PIZZA_ORDER (  
  ID integer generated by default as identity (start with 1),  
3  SIZE_ID integer not null,
```

⁸<http://code.google.com/p/youeat/>

```
ROOM_NUMBER integer not null,  
5 DAY integer not null,  
STATUS integer not null,  
7 primary key (ID)  
);  
9  
create table SYS_TIME (  
11 ID integer not null,  
CURRENT_DAY integer not null,  
13 LAST_REPORT integer not null,  
primary key (ID)  
15 );
```

Listing 4.5: Pizza project database schema.

Table 4.2: Effectiveness of domain model class filtering between RengLaDom and Womble. Cl: Subject Java classes and interfaces. DC: Cl that are domain model classes. Rtr: Cl RengLaDom labeled as domain model classes. Pr: Precision. RC: Recall. RT: Time taken. Average: Average Pr and RC of Womble output. Intersection: Pr and RC of the intersection of Womble outputs. Union: Pr and RC of the union of Womble outputs. Group 1: projects without database. Group 2: projects using JDBC. Group 3: projects using iBATIS. Group 4: projects using MyBatis. Group 5: projects using Hibernate.

Name	Subject Info			RengLaDom						Womble					
	kLOC	CL	DC	Time	Pr	RC	Average	Total	Pr	RC	Average	Intersection	Union		
pdf-sam 2.2.1	8.8	191	9	2'38s	0.38	0.89	29.94s	4'29.5s	0.45	0.14	0.45	1	0.11	0.16	1
pizza_(1.4)	1.1	18	4	36s	1	1	0.06s	0.26s	0.52	0.31	0.52	0.5	0.25	0.57	1
pizza_wo	1.1	18	4	30s	1	1	0.17s	0.69s	0.52	0.31	0.52	0.5	0.25	0.57	1
SweetHome3D 3.3	45.8	1366	40	9'49s	0.33	0.45	-	-	-	-	-	-	-	-	-
janwiki 1.0.5	18.6	212	24	4'14s	0.62	0.88	-	-	-	-	-	-	-	-	-
jforum 2	22.1	397	40	8'1s	0.62	0.78	-	-	-	-	-	-	-	-	-
jpetstore 4.0.5	1.6	42	10	48s	1	1	0.11s	0.95s	0.36	0.21	0.36	1	0.11	0.35	1
activiti 5.6	76.3	2146	32	25'6s	0.13	0.94	-	-	-	-	-	-	-	-	-
alfresco 3.4	647.2	10883	87	15h52'	0.08	0.83	-	-	-	-	-	-	-	-	-
jpetstore 6.0.1	1.1	24	9	1'7s	0.75	0.67	0.07s	0.64s	0.35	0.25	0.35	1	0.11	0.35	1
connect 1.1.2	11.3	279	40	3'5s	0.9	0.9	0.08s	3.02s	0.32	0.13	0.32	NaN	0	0.73	1
JContact 1.0	1.4	40	6	1'12s	0.67	1	0.08s	0.49s	0.31	0.31	0.31	1	0.17	0.46	1
jforum 3	22.9	898	25	7'4s	0.62	0.96	1.54s	38.44s	0.36	0.2	0.36	1	0.04	0.5	1
myblog 1.8.3	6.7	150	5	27s	1	1	0.35s	1.74s	0.31	0.44	0.31	NaN	0	0.36	1
pizza_hib	1.2	19	4	31s	1	1	0.54s	2.16s	0.52	0.31	0.52	0.5	0.25	0.57	1
Q&A System 1.0	25	214	7	4'20s	0.27	1	0.99s	6.91s	0.25	0.41	0.25	NaN	0	0.44	1
Shopizer 1.1.4	53.3	684	76	19'6s	0.43	0.97	-	-	-	-	-	-	-	-	-
xplanner-plus 1.0	44.6	895	17	14'46s	0.15	1	-	-	-	-	-	-	-	-	-
YouEat 1.9	16.5	559	22	4'23s	0.54	0.86	0.98s	21.63s	0.48	0.13	0.48	1	0.05	0.61	1

4.7 Related Work

Todo: related papers to read and discuss in the related work section:

Womble [24]: Very closely related, they also take Java code and infer domain model and multiplicity constraints, but they use a different technique to infer multiplicity constraints. Inferring even more constraints in addition to multiplicity [87, 88].

class2uml is another closely related tool. It reverse engineers Java classes to UML class diagrams. class2uml makes a different trade-off and deliberately avoids complex inter-procedural analysis [25].

Overview of software architecture reconstruction [89].

Relate object-oriented code with an existing relational database [90].

Reverse engineering C++ code to UML class diagrams, including multiplicity constraints [75].

Another way to infer multiplicity constraints [91].

Following may be less closely related, but still close enough to briefly summarize them, as many of them seem to be well-known in the community, as they have been cited frequently.

Desire [92]

Kazman [93]

Tonella [94]

Perry and Wolf [95]

Harris et al. [96, 97]

Murphy and Notkin [98, 99]

Milanova citemilanova05preciseproposes ownership model to to indentify composition relationships for UML classes diagrams. They build an approximate object graph which displays access relationship between runtimes objects: a class - its fields or array fields, a class - local variables of its methods. On the object graph, they de-

temine objects that a particular reference variable or field may point to and determine a boundary subgraph rooted “o” of each object “o”. The edges of boundary subgraph is used to indentify composition relationships. The tools locally analyze fields and methods of a class to determine class relations. RengLaDom can go a step future to determine the relation between 2 classes whose objects are used in a method of a third classes.

Kang [91]proposes method to reverse engineering 1-n multiplicity relation of between classes. First, they analyze java byte code constructs to extract alias relationships: variable - variable, container - container, and container - variable. Second, an abstract heap structure, an approximation of runtime heap structure between container and its elements, is constructed. Relationship, variable - variable and container - container, is used to approximate the runtime relationship of variables which refers to the same object and also the type of the object. The container-variable relationship is used to infer the 1-n relationship. The authors bases on byte code constructs to extracts alias and container relations, but not high level relations between variable objects. RengLaDom not only captures alias relations: variable assignment, return of invoked method call but also high level relations: variables in the same statement to infer domain concept relations.

Sutton and Maletic [75] develop a tools called pilfer to reverse engineering UML classes diagram from C++ source code. Pilfer parses C++ source code to XML format. A translation unit (a unit xml tag) is created for each cpp file and pilfer creates a abstract syntax graph for these units. It perform a lightweight semantic analysis on the abstract syntax graph such as: design level of attributes, correctness of ‘has-a’ class associations, multiplicity of class associations. Finally, pilfer outputs the result as UML class diagram or XMI format. Pilfer is successful works on producing complete and more meaningful UML classes diagram. However, it does not focus on

eliminating “plumbing” classes from set of UML classes. In addition, pilfer separately analyze UML classes, so it fails to detect complex class associations. RengLaDom propose a new technique to eliminate “plumbing” classes and reverse engineering both “has-a” and complex class associations.

Rigi [74] can reverse engineer software (written in C, C++ and Cobol) to visualization form as graphs. Rigi analyzes source code artifact, create a intermediate format, Rigi Standard Format files, and transfers these files to graph models. Rigi can be applied to large legacy systems (million lines of code). However, its output graph is raw with thousand nodes, and a meaningful graph model cannot be done automatically and required interactions of software engineers with some experience.

Ratiu [100] states similar challenges when reverse engineering domain model from source code. Base on commonalities of APIs in the same domain, domain ontology fragments is obtained. Then Ratiu uses some tools to extracts domain term from source code artifact, and manually reverses engineer domain model. RengLaDom is the first tools which addresses these challenges automatically and shows that our technique can be applied for difference APIs.

Daikon [30].

Linux case study [101]

Web applications [102]

Using points-to-analysis [103]

Static technique to locate concepts in object-oriented code [104]. A concept can be a domain model class, but there are also other concepts that are not implemented in a single class.

Re-documentation of Java with UML class diagrams [105]. Is closely related. But they do not analyze methods for complex relations.

Future direction: Visualization of reverse engineering results [106].

4.8 Conclusions and Future Work

With system have database with non-ORM, RengLaDom can combine with current existing Hibernate domain model classes reverse engineering tool such as Hibernate Tools for Eclipse, to analyze the consistense of source code with generated domain model classes of existing tools.

REFERENCES

- [1] G. C. Murphy, M. Kersten, and L. Findlater, “How are Java software developers using the Eclipse IDE?” *IEEE Software*, vol. 23, no. 4, pp. 76–83, July 2006.
- [2] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, “How do professional developers comprehend software?” in *Proc. 34th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, June 2012, pp. 255–265.
- [3] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich, “Touchdevelop: Programming cloud-connected mobile devices via touchscreen,” in *Proc. 10th SIGPLAN ONWARD*. ACM, 2011, pp. 49–60.
- [4] A. Zeller and D. Lütkehaus, “DDD - A free graphical front-end for Unix debuggers,” *SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, Jan. 1996.
- [5] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, Sept. 2008.
- [6] G. Nudelman, *Android Design Patterns: Interaction Design Solutions for Developers*. Wiley, Mar. 2013.
- [7] S. E. Salamati Taba, I. Keivanloo, Y. Zou, J. Ng, and T. Ng, “An exploratory study on the relation between user interface complexity and the perceived quality of Android applications,” in *Proc. 14th International Conference on Web Engineering (ICWE)*. Springer, July 2014.
- [8] Z. Hussain, M. Lechner, H. Milchrahm, S. Shahzad, W. Slany, M. Umgeher, T. Vlk, and P. Wolkerstorfer, “User interface design for a mobile multimedia application: An iterative approach,” in *Proc. 1st International Conference on*

- Advances in Computer-Human Interaction (ACHI)*. IEEE, Feb. 2008, pp. 189–194.
- [9] T. S. da Silva, A. Martin, F. Maurer, and M. S. Silveira, “User-centered design and agile methods: A systematic review,” in *Proc. Agile Conference (AGILE)*. IEEE, Aug. 2011, pp. 77–86.
- [10] K. Kuusinen and T. Mikkonen, “Designing user experience for mobile apps: Long-term product owner perspective,” in *Proc. 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2013, pp. 535–540.
- [11] C. Zeidler, C. Lutteroth, W. Stürzlinger, and G. Weber, “The Auckland layout editor: An improved GUI layout specification process,” in *Proc. 26th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2013, pp. 343–352.
- [12] —, “Evaluating direct manipulation operations for constraint-based layout,” in *Proc. 14th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, Sept. 2013, pp. 513–529.
- [13] The Nielsen Company, “The mobile consumer: A global snapshot,” <http://www.nielsen.com/us/en/insights/reports/2013/mobile-consumer-report-february-2013.html>, Feb. 2013.
- [14] A. K. Karlson, B. Meyers, A. Jacobs, P. Johns, and S. K. Kane, “Working overtime: Patterns of smartphone and PC usage in the day of an information worker,” in *Proc. 7th International Conference on Pervasive Computing (Pervasive)*. Springer, May 2009, pp. 398–405.
- [15] P. Bao, J. S. Pierce, S. Whittaker, and S. Zhai, “Smart phone use by non-mobile business users,” in *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, Aug. 2011, pp. 445–454.

- [16] L. Fortunati and S. Taipale, “The advanced use of mobile phones in five European countries,” *The British Journal of Sociology*, vol. 65, no. 2, pp. 317–337, June 2014.
- [17] E. J. Chikofsky and J. H. C. II, “Reverse engineering and design recovery: A taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990.
- [18] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, “Reverse engineering: A roadmap,” in *Proc. Conference on The Future of Software Engineering*. ACM, June 2000, pp. 47–60.
- [19] R. Farenhorst, J. F. Hoorn, P. Lago, and H. van Vliet, “The lonesome architect,” in *Proc. Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture (WICSA/ECSA)*. IEEE, Sept. 2009, pp. 61–70.
- [20] A. Forward and T. C. Lethbridge, “Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals,” in *Proc. International Workshop on Models in Software Engineering (MiSE)*. ACM, May 2008, pp. 27–32.
- [21] B. Anda and K. Hansen, “A case study on the application of UML in legacy development,” in *Proc. 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*. ACM, Sept. 2006, pp. 124–133.
- [22] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, “The impact of UML documentation on software maintenance: An experimental evaluation,” *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 6, pp. 365–381, June 2006.
- [23] W. J. Dzidek, E. Arisholm, and L. C. Briand, “A realistic empirical evaluation of the costs and benefits of UML in software maintenance,” *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 3, pp. 407–432, May 2008.

- [24] D. Jackson and A. Waingold, “Lightweight extraction of object models from bytecode,” in *Proc. 21st ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 1999, pp. 194–202.
- [25] M. Keschenau, “Reverse engineering of UML specifications from Java programs,” in *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2004, pp. 326–327.
- [26] R. M. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 10th ed. Free Software Foundation, Feb. 2011.
- [27] C.-y. Chao, “Why can’t smart phones be polite, too? What would a phone need to know?” Ph.D. dissertation, Massachusetts Institute of Technology, Feb. 2011.
- [28] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, “Debug all your code: portable mixed-environment debugging,” in *Proc. 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2009, pp. 207–226.
- [29] A. J. Ko and B. A. Myers, “Designing the Whyline: A debugging interface for asking questions about program behavior,” in *Proc. ACM SIGCHI CHI*. ACM, Apr. 2004, pp. 151–158.
- [30] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
- [31] Y. Brun and M. D. Ernst, “Finding latent code errors via machine learning over program executions,” in *Proc. 26th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, May 2004, pp. 480–490.

- [32] C. Csallner and Y. Smaragdakis, “Dynamically discovering likely interface invariants,” in *Proc. 28th ACM/IEEE International Conference on Software Engineering (ICSE), Emerging Results Track*. ACM, May 2006, pp. 861–864.
- [33] C. Csallner, N. Tillmann, and Y. Smaragdakis, “DySy: Dynamic symbolic execution for invariant inference,” in *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 2008, pp. 281–290.
- [34] Ø. D. Trier, A. K. Jain, and T. Taxt, “Feature extraction methods for character recognition—a survey,” *Pattern Recognition*, vol. 29, no. 4, pp. 641–662, Apr. 1996.
- [35] R. Plamondon and S. Srihari, “Online and off-line handwriting recognition: A comprehensive survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 63–84, Jan. 2000.
- [36] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. O’Reilly, Oct. 2008.
- [37] B. A. Myers, “Graphical user interface programming,” in *Computer Science Handbook*, 2nd ed., A. B. Tucker, Ed. CRC Press, May 2012.
- [38] Apple Inc., “View programming guide for iOS,” https://developer.apple.com/library/ios/documentation/windowsviews/conceptual/viewpg_iphoneos/ViewPG_iPhoneOS.pdf, Oct. 2013, accessed May 2015.
- [39] V. Nahavandipoor, *iOS 7 Programming Cookbook*, 1st ed. O’Reilly, Nov. 2013.
- [40] M. Gargenta and M. Nakamura, *Learning Android: Develop Mobile Apps Using Java and Eclipse*, 2nd ed. O’Reilly, Jan. 2014.
- [41] B. A. Myers, “Challenges of HCI design and implementation,” *Interactions*, vol. 1, no. 1, pp. 73–83, Jan. 1994.

- [42] A. S. Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder, “Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps,” in *Proc. ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS)*. ACM, June 2013, pp. 275–284.
- [43] D. Karatzas, F. Shafait, S. Uchida, M. Iwamura, L. G. i Bigorda, S. R. Mestre, J. Mas, D. F. Mota, J. Almazàn, and L. de las Heras, “ICDAR 2013 robust reading competition,” in *Proc. 12th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Aug. 2013, pp. 1484–1493.
- [44] R. Smith, “An overview of the Tesseract OCR engine,” in *Proc. 9th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Sept. 2007, pp. 629–633.
- [45] R. W. Smith, “Hybrid page layout analysis via tab-stop detection.” in *Proc. 10th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, July 2009, pp. 241–245.
- [46] A. Antonacopoulos, S. Pletschacher, D. Bridson, and C. Papadopoulos, “ICDAR 2009 page segmentation competition,” in *Proc. 10th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, July 2009, pp. 1370–1374.
- [47] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer, Nov. 2010.
- [48] R. Smith, “An overview of the tesseract ocr engine,” in *ICDAR ’07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 629–633. [Online]. Available: <http://www.google.de/research/pubs/archive/33418.pdf>

- [49] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, Nov. 1986.
- [50] T. A. Nguyen, C. Csallner, and N. Tillmann, “GROPG: A graphical on-phone debugger,” in *Proc. 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track*. IEEE, May 2013, pp. 1189–1192.
- [51] Jay Freeman, “Cycrypt,” <http://www.cycrypt.org/>, 2014, accessed May 2015.
- [52] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, Feb. 1966.
- [53] M. W. Newman and J. A. Landay, “Sitemaps, storyboards, and specifications: A sketch of Web site design practice as manifested through artifacts,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-99-1062, 1999.
- [54] Y. Y. Wong, “Rough and ready prototypes: Lessons from graphic design,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), Posters and Short Talks*. ACM, 1992, pp. 83–84.
- [55] J. A. Landay and B. A. Myers, “Interactive sketching for the early stages of user interface design,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, May 1995, pp. 43–50.
- [56] P. F. Campos and N. J. Nunes, “Practitioner tools and workstyles for user-interface design.” *IEEE Software*, vol. 24, no. 1, pp. 73–80, Jan. 2007.
- [57] B. A. Myers, S. E. Hudson, and R. F. Pausch, “Past, present, and future of user interface software tools,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 1, pp. 3–28, Mar. 2000.

- [58] S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. P. Mertz, “Revisiting visual interface programming: creating GUI tools for designers and programmers,” in *Proc. 17th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2004, pp. 267–276.
- [59] J. A. Landay and B. A. Myers, “Sketching interfaces: Toward more human interface design,” *IEEE Computer*, vol. 34, no. 3, pp. 56–64, Mar. 2001.
- [60] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge, “JavaSketchIt: Issues in sketching the look of user interfaces,” in *Proc. AAAI Spring Symposium on Sketch Understanding*. AAAI, Mar. 2002, pp. 9–14.
- [61] A. Coyette, S. Kieffer, and J. Vanderdonckt, “Multi-fidelity prototyping of user interfaces.” in *Proc. 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, Sept. 2007, pp. 150–164.
- [62] 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 *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, Aug. 2011, pp. 109–112.
- [63] M. Dixon and J. Fogarty, “Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Apr. 2010, pp. 1525–1534.
- [64] M. Dixon, D. Leventhal, and J. Fogarty, “Content and hierarchy in pixel-based methods for reverse engineering interface structure,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, May 2011, pp. 969–978.
- [65] T.-H. Chang, T. Yeh, and R. C. Miller, “Associating the visual representation of user interfaces with their internal structures and metadata,” in *Proc. 24th*

- Annual ACM Symposium on User Interface Software and Technology (UIST)*.
ACM, Oct. 2011, pp. 245–256.
- [66] X. Meng, S. Zhao, Y. Huang, Z. Zhang, J. Eagan, and R. Subramanian, “WADE: simplified GUI add-on development for third-party software,” in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Apr. 2014, pp. 2221–2230.
- [67] M. de Sà, L. Carriço, L. Duarte, and T. Reis, “A mixed-fidelity prototyping tool for mobile devices,” in *Proc. Working Conference on Advanced Visual Interfaces (AVI)*. ACM, May 2008, pp. 225–232.
- [68] S. P. Reiss, “Seeking the user interface,” in *Proc. 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, Sept. 2014, pp. 103–114.
- [69] A. Martínez, H. Estrada, J. Sánchez, and O. Pastor, “From early requirements to user interface prototyping: A methodological approach,” in *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Sept. 2002, pp. 257–260.
- [70] M. Book and V. Gruhn, “Modeling web-based dialog flows for automatic dialog control,” in *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Sept. 2004, pp. 100–109.
- [71] J. Falb, T. Röck, and E. Arnautovic, “Using communicative acts in interaction design specifications for automated synthesis of user interfaces,” in *Proc. 21st ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2006, pp. 261–264.
- [72] A. Khambati, J. C. Grundy, J. Warren, and J. G. Hosking, “Model-driven development of mobile personal health care applications,” in *Proc. 23rd ACM/IEEE*

- International Conference on Automated Software Engineering (ASE)*. IEEE, Sept. 2008, pp. 467–470.
- [73] S. Barnett, R. Vasa, and J. Grundy, “Bootstrapping mobile app development,” in *Proc. 37th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, May 2015.
- [74] H. M. Kienle and H. A. Müller, “Rigi-an environment for software reverse engineering, exploration, visualization, and redocumentation,” *Science of Computer Programming*, vol. 75, no. 4, pp. 247–263, Apr. 2010.
- [75] A. Sutton and J. I. Maletic, “Recovering UML class models from C++: A detailed explanation,” *Information and Software Technology*, vol. 49, no. 3, pp. 212–229, Mar. 2007.
- [76] M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Oct. 1996.
- [77] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall, Oct. 2004.
- [78] E. J. O’Neil, “Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM),” in *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, June 2008, pp. 1351–1356.
- [79] C. Bauer and G. King, *Java persistence with Hibernate*. Manning, Nov. 2006.
- [80] P. P. Chen, “The entity-relationship model—toward a unified view of data,” *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976.
- [81] M. Flower, *Patterns of Enterprise Application Architecture*. Pearson Education, 2003.
- [82] S. Kearney and J. F. Power, “Benchmarking the accuracy of reverse engineering tools for Java programs: A study of eleven UML tools,” Dept. of Computer

- Science, National University of Ireland Maynooth, Tech. Rep. NUIM-CS-TR-2007-01, Jan. 2007.
- [83] L. C. Briand, “The experimental paradigm in reverse engineering: Role, challenges, and limitations,” in *Proc. 13th Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2006, pp. 3–8.
- [84] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, “Empirical studies in reverse engineering: State of the art and future trends,” *Empirical Software Engineering*, vol. 12, no. 5, pp. 551–571, Oct. 2007.
- [85] Y.-G. Guéhéneuc, K. Mens, and R. Wuyts, “A comparative framework for design recovery tools,” in *Proc. 10th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, Mar. 2006, pp. 123–134.
- [86] A. Sorensen and A. R. Brown, “Introducing jMusic,” in *Proc. Australasian Computer Music Conference (ACMC)*. Australasian Computer Music Association (ACMA), July 2000, pp. 68–76.
- [87] Y.-G. Guéhéneuc and H. Albin-Amiot, “Recovering binary class relationships: Putting icing on the UML cake,” in *Proc. 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2004, pp. 301–314.
- [88] Y.-G. Guéhéneuc, “A reverse engineering tool for precise class diagrams,” in *Proc. Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, Oct. 2004, pp. 28–41.
- [89] S. Ducasse and D. Pollet, “Software architecture reconstruction: A process-oriented taxonomy,” *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 4, pp. 573–591, July 2009.

- [90] C. Marinescu, “Discovering the objectual meaning of foreign key constraints in enterprise applications,” in *Proc. 14th Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2007, pp. 100–109.
- [91] Y. Kang, C. Park, and C. Wu, “Reverse-engineering 1-n associations from Java bytecode using alias analysis,” *Information and Software Technology*, vol. 49, no. 2, pp. 81–98, Feb. 2007.
- [92] T. J. Biggerstaff, “Design recovery for maintenance and reuse,” *Computer*, vol. 22, no. 7, pp. 36–49, July 1989.
- [93] R. Kazman and S. J. Carrière, “Playing detective: reconstructing software architecture from available evidence,” *Automated Software Engineering*, vol. 6, no. 2, pp. 107–138, Apr. 1999.
- [94] P. Tonella, “Reverse engineering of object oriented code,” in *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 2005, pp. 724–725.
- [95] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.
- [96] D. R. Harris, H. B. Reubenstein, and A. S. Yeh, “Reverse engineering to the architectural level,” in *Proc. 17th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, Apr. 1995, pp. 186–195.
- [97] D. R. Harris, A. S. Yeh, and H. B. Reubenstein, “Recognizers for extracting architectural features from source code,” in *Proc. 2nd Working Conference on Reverse Engineering (WCRE)*. IEEE, July 1995, pp. 252–261.
- [98] G. C. Murphy and D. Notkin, “Reengineering with reflexion models: A case study,” *Computer*, vol. 30, no. 8, pp. 29–36, Aug. 1997.

- [99] G. C. Murphy, D. Notkin, and K. J. Sullivan, “Software reflexion models: Bridging the gap between design and implementation,” *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 4, Apr. 2001.
- [100] D. Ratiu, “Reverse engineering domain models from source code,” *Reverse Engineering Models from Software Artifacts REM 2009*, p. 13, 2009.
- [101] I. T. Bowman, R. C. Holt, and N. V. Brewster, “Linux as a case study: Its extracted software architecture,” in *Proc. 21st ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 1999, pp. 555–563.
- [102] A. E. Hassan and R. C. Holt, “Architecture recovery of web applications,” in *Proc. 24th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 2002, pp. 349–359.
- [103] A. Milanova, “Precise identification of composition relationships for UML class diagrams,” in *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Nov. 2005, pp. 76–85.
- [104] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, “Static techniques for concept location in object-oriented code,” in *Proc. 13th International Workshop on Program Comprehension (IWPC)*. IEEE, May 2005, pp. 33–42.
- [105] M. Gogolla and R. Kollmann, “Re-documentation of Java with UML class diagrams,” in *Proc. 7th Reengineering Forum, Reengineering Week*, Mar. 2000, pp. 41–48.
- [106] M. Lanza and S. Ducasse, “Polymetric views—a lightweight visual approach to reverse engineering,” *IEEE Transactions on Software Engineering (TSE)*, vol. 29, no. 9, pp. 782–795, Sept. 2003.

BIOGRAPHICAL STATEMENT

Tuan Anh Nguyen was born in Vietnam, in 1984. He received his B.S. degree from University of Natural Science, Vietnam, in 2007, his M.S. degree from The University of Texas at Arlington (UTA) in 2012. His current research interest is in mobile software engineering, especially in reverse engineering and program analysis. During his PhD program at UTA, he worked for Google.