### **Software Analytics for Mobile Applications**

Master's Thesis submitted to the Faculty of Informatics of the *Università della Svizzera Italiana* in partial fulfillment of the requirements for the degree of Master of Science in Informatics Software Design

presented by
Roberto Minelli

under the supervision of Prof. Dr. Michele Lanza co-supervised by Dr. Marco D'Ambros

June 2012

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

> Roberto Minelli Lugano, 22 June 2012

"Education is what remains after one has forgotten everything he learned in school". — Albert Einstein

# Abstract

"App" is a term used to describe software that runs on smartphones and other handheld devices. Apps vary in theme from games to music players to productivity tools. Apps development gained momentum starting in 2007, when Apple launched its first iPhone. Nowadays the impact of mobile applications in the software industry is significant, considering that nearly one million apps are available from the Apple and Android stores.

We argue that apps are different from traditional software systems in many ways: from source code structure to development methodologies adopted. Since the first apps were developed only a few years ago, they have not been thoroughly investigated until now.

To this aim, we propose to study how apps development differs from classical software development focusing our analysis on three factors: source code, usage of third-party APIs, and evolution. To support the analysis, we implemented *SAMOA*: A tool that extracts and visualizes software facts from the source code of apps. Our tool provides a catalogue of views, at ecosystem or single-app granularity, to examine apps from different perspectives. In addition *SAMOA* lets the user navigate the history of apps to inspect how they evolved over time.

We used *SAMOA* to investigate a selection of open-source Android apps. Our tool helped us to discover unique characteristics of apps. We summarized our findings in a catalogue presented in this thesis. <u>vi</u>\_\_\_\_\_

# Acknowledgements

First of all I would like to thank Prof. Dr. Michele Lanza, one of my favorite and most appreciated professors, in the capacity of supervisor of my Master Thesis. I am grateful for the opportunity to work within the *REVEAL* research group on this project. I will always treasure the knowledge and passion for work that you shared with me.

Many thanks to Dr. Marco D'Ambros, the co-advisor of the project, that proposes me this work and continuously stimulates my enthusiasm with new ideas. Thanks also for the advices and feedbacks during the writing of the thesis.

To all members of the *REVEAL* group of the University of Lugano: thank you! Being part of your group and sharing chats with you was a great experience. Thank you all for providing me support when needed.

Special thanks to my mother Gabriella and my father Sergio for their unconditional support, love, patience and for always believing in me. I understand how hard is living with me when I'm under pressure but I know you already forgiven me. Thank you!

To all my friends, thanks for being who you are and always believing in me. I know I have not showed up much in the last year, but I'll catch up with you this Summer, I promise.

Roberto Minelli June 12, 2012 viii

# Contents

Co	nten	ts			ix			
Fig	gures				xi			
Ta	Tables xiii							
1	<b>Intro</b> 1.1 1.2		<b>on</b> ibutions		1 3 3			
2	2.1 2.2 2.3	What is Reverse Model 2.3.1 2.3.2	o: The Era of Mobile Apps is an App?		<b>5</b> 6 7 8 8 9 9			
3	Rela 3.1 3.2 3.3 3.4 3.5	Mining Softwa Softwa	ork le Applications Studies	· · ·	<ol> <li>13</li> <li>14</li> <li>15</li> <li>16</li> <li>17</li> </ol>			
4	<ul> <li>SAM</li> <li>4.1</li> <li>4.2</li> <li>4.3</li> <li>4.4</li> </ul>	Archite Visuali 4.2.1 4.2.2 4.2.3 User In	tecture Overview		<ol> <li>19</li> <li>19</li> <li>22</li> <li>22</li> <li>26</li> <li>29</li> <li>30</li> <li>31</li> <li>33</li> </ol>			

		4.4.3 Grid View Interactions	33
		4.4.4 Bar Chart Interactions	34
		4.4.5 Line chart Interactions	35
	4.5	Metrics	35
	4.6	Summing Up	37
F		atalague of Characteristics of Mahila Applications	39
5	5.1	atalogue of Characteristics of Mobile Applications         Snapshot-based	<b>39</b> 39
	5.1	5.1.1 Dominance of External Calls	39 40
		5.1.2 Dominance of Internal Calls	40 41
		5.1.2 Dominance of internal Calls	41 42
		5.1.4 High Core Ratio	43
		5.1.5 Low Core Ratio	44
		5.1.6 Multiple Main Activities	46
	52	History-based	47
	5.2	5.2.1 Core Drop	48
		5.2.2 Delayed Use of Versioning Systems	49
		5.2.3 Flat Intervals in History	50
		5.2.4 Gradual Increase of Core Elements	51
		5.2.5 High Correlation Between LOC and Third-Party Calls	52
		5.2.6 Out-of-Sync Manifest File	54
		5.2.7 Snap Decrease of Core Elements	55
		5.2.8 Snap Increase of Core Elements	57
		5.2.9 Stepwise Growth in History	58
		5.2.10 Stepwise Increase of Core Elements	59
	5.3	Summing Up	60
_	-		
6		clusions	61
	6.1		61
		Limitations	62
	6.3	Future Work     Image: State of the state of	63
		6.3.1 Support for Additional Platforms	63
		6.3.2       Support for Additional Versioning Systems         6.3.3       Online Service	63 63
		<ul><li>6.3.3 Online Service</li></ul>	63 64
		6.3.5 Enhancements to the Analysis	64
		6.3.6       Enrichments of the Catalogue	64
		6.3.7       Validation with a Larger Dataset	64
	64	Epilogue	65
	0.4		05
Α	SAN	IOA: Implementation Details	67
	A.1	Data collection & export	67
	A.2	Data retrieval & visualization	71
	A.3	Summing Up	72
В	Add	litional Details on the Dataset	73
<b></b>	1. 1•		
Bi	bliog	raphy	75

# Figures

2.1	An example of the Overview Pyramid applied to ArgoUML	6
2.2	Quickdroid's Overview Pyramid	7
2.3	The template to describe mobile apps	8
2.4	A screenshot of the F-Droid client for Android phone	9
4.1	Architectural overview of SAMOA	19
4.2	A first glance of <i>SAMOA</i> visualizing the MythMote application.	21
4.3	Single-app visualization and color legend	23
4.4	The proportions in the single-app view	24
4.5	A visualization of the distribution of LOC of Open-GPS-Tracker	25
4.6	The same visualization of Figure 4.5 with grouped bars.	25
4.7	A visualization of the distribution of the number of core elements of Zirco-Browser	26
4.8	A grid view of an App Ecosystem	26
4.9	Another grid view of an App Ecosystem	27
4.10	External calls distribution of the app ecosystem.	28
4.11	Number of commits among the app ecosystem	28
4.12	The same visualization of Figure 4.10 with grouped bars	29
4.13	Distribution of LOC sorted in ascending order according to coreLOC.	29
4.14	Single-app view of <i>SAMOA</i> visualizing the Alogcat application at revision 48	31
4.15	SAMOA displaying a snapshot view of AndLess	32
4.16	Navigation possibilities and possible paths offered by the Tools Panel	33
4.17	SAMOA's source code viewer	34
5.1	Dominance of External Calls in Share My Position application.	40
5.2	Dominance of Internal Calls in Replica Island application.	41
5.3	God-class smell in the SearchLight application	42
5.4	High core ratio symptoms in the AndLess application	43
5.5	Low core ratio symptoms in the Android VNC Viewer application.	44
5.6	Low core ratio symptoms in the Sipdroid application.	45
5.7	Multiple main activities in the App SoundManager application	46
5.8	Core drop in the Open GPS Tracker application.	48
5.9	A comparison of the two core snapshots presenting the core drop	48
5.10	Delayed use of SVN in Solitaire for Android application.	49
5.11	Flat intervals in the LOC distribution of Share My Position.	50
5.12	The distribution of the number of core elements of the Zirco Browser application.	51
5.13	The distribution of LOC of the Zxing application	52

5.14 The distribution of third-party calls of the Zxing application.	52
5.15 Out-of-sync manifest file of the Zxing application.	54
5.16 The distribution of the number of core elements of the Csipsimple application.	55
5.17 The distribution of the number of core elements of the VNC Viewer application.	57
5.18 Jumps in the LOC distribution of Apps Organizer application	58
5.19 The distribution of the number of core elements of the Zxing application	59
A.1 The object graph of a simple model	69

# **Tables**

2.1	Criteria for the selection and corresponding apps.	9
2.2	List of all apps analyzed in the preliminary study.	10
2.3	Our App Ecosystem. The dataset used for the detailed investigation.	11
B.1	Links to the websites of apps	73
B.2	Repository URLs of our dataset.	74

## **Chapter 1**

## Introduction

Researchers have studied many aspects of traditional software systems: development methodologies, design patters, structure of the code, evolution, etc. People consider software as a classical engineering product, but it is more complex than many other human artifacts. One of the reasons behind software complexity is that software evolves over time [Leh96]. Evolution makes maintenance one of the activities with the highest impact on the cost of a software system. Sommerville [Som89] and Davis [Dav95] have estimated that software maintenance takes from 50% to 75% of the total cost of a software product. According to Lehman [LPR<sup>+</sup>97] software evolution results from continuous changes and increase in both complexity and size of software. At the same time maintenance increases *software entropy* where architecture and design drift from the original decisions.

One of the goals of software evolution analysis is to identify potential defects in its logic or architecture [Leh85]. This process is often supported by software reverse engineering, aimed at creating higher level representations of the system to identify the system's components and their interrelationships [CI90]. Software visualizations are one of the means to represent software systems. In particular, software evolution visualization applies software visualization techniques to evolutionary data. Those data are typically stored in software configuration management (SCM) systems, thus mining software repositories plays a key role during software evolution analysis.

Mobile applications (a.k.a. apps) are software systems implemented to run on smartphones and other handheld devices. Apps have different genres, from games to utilities, from productivity tools to browsers and are implemented using a variety of programming languages. Apps are, in all respects, software systems but since the first apps were developed only a few years ago, they have not been thoroughly investigated until now.

Researchers — instead — proposed various approaches for software maintenance, software evolution, and program comprehension of traditional software systems [Leh96, LPR<sup>+</sup>97, LPR98, GJKT97, GT00, Tur02]. We claim that apps differ from traditional software systems in many ways. In this thesis we analyze mobile applications, and the way they are developed, to discover the differences. Our research spans multiple domains of software engineering, from software evolution analysis, to mining software repositories, to software visualization.

In 2007 Apple introduced the iPhone and, one year after, the AppStore, *i.e.*, a platform where people publish and sell their mobile applications. This event opened a new world for software developers and mobile applications development has gained momentum. Big software companies (*i.e.*, Adobe, EA) started porting some of their desktop applications to mobile devices. In the following years Google, Microsoft, Nokia and others created their own app stores and entered the competition. Nowadays the impact of apps in the software industry is significant, considering that about one million of them are available from the Apple and Android stores.

According to Islam *et al.* [IIM10] apps development is one of the fastest growing businesses. The authors report that in 2009 about 6.4 billion apps have been downloaded. Roughly 60% of those were free but the remaining 40% generated an overall income of about \$4.5 billion. Markets & Markets estimated that the global apps business is expected to be worth \$25 billion in 2015 [mar10].

Our research focuses on software design aspects rather than marketing. Since the first apps were developed only a few years ago, software design aspects of mobile applications have not been thoroughly investigated so far. In fact, we know little about the code structure of apps or the development methodologies adopted.

In this thesis we argue that apps differ from traditional software systems. For example, we observed that the use of well known object-oriented paradigms (*i.e.*, inheritance) is nearly absent in apps. Apps are, most of the times, single-developer projects that last for a short period with respect to classical software systems developed in large teams and persisting for years.

For these reasons, we propose to study how apps differ from classical software systems. We focus our analysis on three factors: source code metrics, usage of third-party application programming interfaces (APIs), and evolution. We claim that apps make intensive use of third-party APIs. To this aim, we studied distribution of calls to external libraries. We modeled apps as a set of method invocations and a core. We are interested in understanding how this ensemble evolves over time. We analyzed open-source apps using versioning systems (such as SVN<sup>1</sup> and Git<sup>2</sup>) to handle code evolution.

To support our analysis, we implemented *SAMOA*, a web-based tool that extracts and visualizes software facts from both apps' source code and software configuration management (SCM) meta-data. We argue that software visualization is a powerful means to support software analysis. Our tool provides a catalogue of views at two different granularities: ecosystem and single-app. *SAMOA* helps to understand how different aspects of apps evolved over time. The tool provides visualizations showing how values of a set of software metrics evolve over time or means to see the state of a snapshot of an application.

Unfortunately there is no *official repository* of apps' source code and not all apps are open source. For our study, we decided to investigate Java apps (*i.e.*, Android) because are retrievable as open-source projects.

<sup>&</sup>lt;sup>1</sup>See http://svncorp.org <sup>2</sup>See http://git-scm.com/

Our tool helped us to discover events happening during the development of apps that are not common in traditional software development. We collected our findings in a catalogue of characteristics of mobile applications, which have emerged using *SAMOA* to explore apps.

### 1.1 Contributions

The main contributions of this work can be summarized as:

- An extensive software design analysis of mobile applications.
- A template to summarize apps, based on software metrics and apps-specific concepts.
- A catalogue of interactive visualizations, at single-app and ecosystem granularity, which help understanding how apps are developed and maintained.
- The implementation of a software analytics tool customized for apps that is available to others as an interactive web application.
- A catalogue of characteristics of apps, collected using the views provided by our tool.

### 1.2 Structure of the Document

This document is organized as follows:

- **Chapter 1** (p. 1) sets the domain for our thesis. It introduces the main research fields involved, explains the motivation of the work and outlines the document structure.
- **Chapter 2** (p. 5) introduces mobile apps. It details mobile applications in general and extracts the requirements of our tool.
- **Chapter 3** (p. 13) describes the related work and explains how it distinguishes from our work.
- **Chapter 4** (p. 19) illustrates our approach and details the tool we developed to support the analysis of mobile applications, *i.e.*, *SAMOA*.
- **Chapter 5** (p. 39) presents a catalogue of characteristics of apps. The chapter provides examples about features of apps that we discovered using our views.
- **Chapter 6** (p. 61) summarizes and reviews our approach by going over the contributions of the work. It discusses also future work that we plan to do to improve our tool.
- **Appendix A** (p. 67) illustrates some implementation details and issues we faced while developing *SAMOA*.
- **Appendix B** (p. 73) reports details about the location of apps, *i.e.*, their repository locations and their websites.

1.2 Structure of the Document

4

### **Chapter 2**

## Intermezzo: The Era of Mobile Apps

This chapter serves as an introduction to mobile applications. We describe mobile applications (Section 2.1) and then we explain how we reverse engineered mobile apps in Section 2.2. Section 2.3 details how we modeled apps and the final part of the chapter (Section 2.4) describes how we composed the apps ecosystem.

### 2.1 What is an App?

In January 2011, the American Dialect Society<sup>1</sup> named "*app*" the word of the year for 2010. The word app is short for "application", in this case a software application. App typically refers to software developed to run on a specific mobile platform. Such platforms include smartphones, tablets and other enabled devices (*e.g.*, iPod Touch). Each app is a self-contained piece of software with a specific purpose, requirements, and set of functionalities. Apps are implemented in a variety of programming languages. The language used depends on the mobile platform: For example, apps for iPhone are developed in Objective-C<sup>2</sup>, a set of extensions to the standard ANSI C language. Android apps are developed in Java and compiled to Dalvik byte-code<sup>3</sup>. Other programming languages are used to write apps for Blackberry, Nokia, and Windows Phone.

Apps are developed using Software Development Kits (SDK) installed in Integrated Development Environments (IDEs). To implement Android apps, for example, Google provides an Eclipse Plugin. SDKs supply facilities to implement, test, and deploy apps inside IDEs. Most of them feature interactive graphical user interface builders and an emulator, useful to test new applications in a virtual environment.

Our study considers the source code as a crucial information. Apps are available for download from App stores but source code is not present. Section 2.4 explains the approach we adopted to tackle this limitation. The next section describes the reverse engineering activities we made to deduce apps' design and devise a template to depict them.

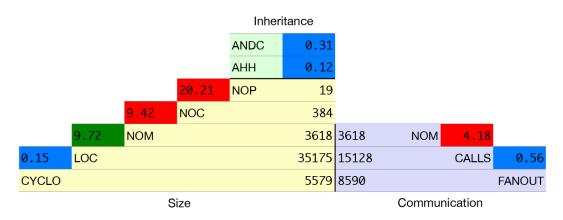
<sup>&</sup>lt;sup>1</sup>See http://www.americandialect.org/

<sup>&</sup>lt;sup>2</sup>See http://developer.apple.com/

<sup>&</sup>lt;sup>3</sup>See http://www.dalvikvm.com/

#### 2.2 Reverse Engineering Apps

Reverse engineering consists in "analyzing a subject system to: identify components and their relationships, and create more abstract representations" [CI90]. Demeyer et al. [DDN00] devised a pattern language that describes how to reverse engineer an object-oriented software system. We applied some of their *First contact patterns* to acquire an initial understanding of mobile apps. For example, we "read all the code in one hour". This approach is useful to get a first impression of the quality of the source code. With this technique we observed that apps were significantly smaller than traditional systems and that, in most of the cases, both test suites and comments were completely absent. By quickly scanning the source code we also hypothesize on which are root classes that define domain abstractions. Other first contact patterns are "skim the documentation" and "do a mock installation". Most applications have a Wiki page that describes the app features and serves as a user guide. We observed that there is not much technical documentation that we could use to support the understanding of the source code of apps. Finally, we tried some applications on the Android Emulator running on top of Eclipse to see how they look like and what we could expect from such mobile applications.



**Figure 2.1.** An example of the Overview Pyramid applied to ArgoUML. BLUE means a low value; GREEN means a value close to average; RED represents a high value.

Besides first contact patters, researchers offered means to characterize the design of a system. Lanza & Marinescu [LM06], for example, proposed the *Overview Pyramid*: "a means to characterize a system from three different viewpoints: size and structural complexity, coupling, and the usage of inheritance". Figure 2.1 shows an overview pyramid applied to ArgoUML<sup>4</sup>.

The authors provide statistical thresholds to determine if values are low, high or close to average (computed on 45 Java and 37 C++ traditional software systems). We were inspired by their idea and we tried this technique on some apps. Our aim was to verify whether the thresholds, provided by Lanza & Marinescu, apply also to apps. Figure 2.2 depicts an Android application — Quickdroid<sup>5</sup> — using the Overview Pyramid. The first thing we noticed, was the nearly total absence of inheritance. The app presented in the figure is one of the few that make use of inheritance. For some apps values were above the high threshold while in other case

<sup>&</sup>lt;sup>4</sup>See http://argouml.tigris.org/

<sup>&</sup>lt;sup>5</sup>See http://code.google.com/p/quickdroid/

were strictly below average. By studying the distribution of the metrics used in the Overview Pyramid (described in Section 4.5) we discovered some interesting correlations. For example, we noticed a high correlation between the number of external calls and the McCabe Cyclomatic complexity number (*i.e.*, 0.82) [McC76]. The correlation of lines of code (LOC) and number of external calls is high as well (*i.e.*, 0.84). This means that the size and complexity of apps grow together with the addition of external method calls.

				ANDC	0.43			
				AHH	0.1			
			4.86	NOP	7			
		8.06	NOC		34			
	8.67	NOM			274	274 NOM	3.85	
0.27	LOC				2375	1056	CALLS	0.45
CYCLO					646	479		FANOUT

**Figure 2.2.** Quickdroid's Overview Pyramid. Colors are used to interpret the pyramid. BLUE means a low value; GREEN means a value close to average; and RED represents an high value.

Colors in the Figure are used to interpret the pyramid according to the reference values proposed by Lanza & Marinescu [LM06]. The Figure shows that some of the values are close to average, but others are significantly lower (or higher) than average. For example, Quickdroid has very few classes per package with respect to the statistical average threshold, that is 17. The inheritance height (AHH) is low while the *coupling intensity* (*i.e.*, CALLS/NOM) is very high.

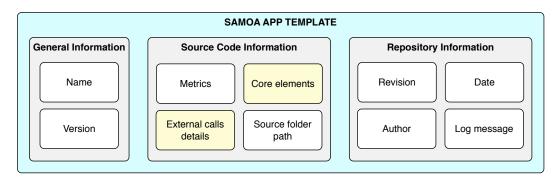
We built other pyramids obtaining results distant from the reference values for traditional software systems. This brought us to claim that apps were different systems, with respect to classical ones. After this preliminary analysis we devised a means to model mobile applications, described in Section 2.3.

#### 2.3 Modeling Apps

For our analysis we devised a language-independent template to describe mobile applications. In this section we present the template and motivate our choices. The template is based on source code and considers both classical software metrics and concepts specific to mobile applications. Figure 2.3 depicts the different aspects of our template. The rest of the section illustrates the most important concepts: the core of the app and the external calls, highlighted in the Figure.

#### 2.3.1 The Core of an App

Apps are software projects composed of source files and other resources, such as XML files containing properties of apps. The source code of apps contains classes that are specific to apps' development (*i.e.*, inherits from the mobile platform SDK) and others generic classes. For this reason, we distinguish two sets of classes, defining as "core of the app" the classes that are



**Figure 2.3.** The template to describe mobile apps. Highlighted in the picture the two key elements, the core elements and the external calls distribution

specific to apps' development. In Android projects, for example, a developer has to implement at least one Activity<sup>6</sup>. An Activity is a single, focused task that the user can do in the application. An Android application without an Activity cannot exist. Activities are declared in a file, the Android Manifest, a file that every application must have. It presents essential information about the application to the Android system. We used the information contained in the manifest to extract the core of the apps. We classified as core all the Activities and Services. Services are similar to Activities and consist of long-running operations that require no interaction with the user. We studied the distribution of lines of codes of the core (coreLOC) with respect to the total number of lines of code (LOC) and we extract relationships among core elements.

#### 2.3.2 The Third-Party Calls

Apps intensively rely on external libraries. We observed that both the size of apps and their complexity grow together with the addition of external method invocations. With this in mind, we studied the distribution of the method calls within applications. We distinguished between calls which implement internal behaviors (INTC) and calls that refers to external libraries (EXTC).

### 2.4 The Apps Ecosystem

Apps are available for download from app stores. These are centralized places which aggregate different pieces of information. On most of them users can rate apps and submit feedbacks and comments. App stores track also the number of downloads of each app to provide different rankings. The main limitation is that stores do not provide source code, as also pointed out by Harman *et al.* [HJZ12]. For this reason, we had to find an alternative place to obtain apps' source code. We use F-Droid<sup>7</sup>, a catalogue of Free and Open Source (FOSS) applications for the Android platform. Figure 2.4 shows a screenshot of the F-Droid client for Android. The following sections detail the approach we followed.

8

<sup>&</sup>lt;sup>6</sup>See http://developer.android.com/ for more details on the Android platform <sup>7</sup>See http://f-droid.org/



Figure 2.4. A screenshot of the F-Droid client for Android phone

#### 2.4.1 Crawling F-Droid

The first goal was to discover how many apps were stored in this repository and how many of them come with source code. To this aim, we built a web crawler. We collected statistics for about 120 apps having either revision information, bug-related information, or both. On average apps have about 400 revisions and 160 issues listed on their bug-tracker page. The median of the revisions' distribution is 98 (the range varies from 3 to 4123). We decided to continue our analysis using this repository given the significant amount of data. For the sake of simplicity we only considered apps using SVN repositories on Google code. We had to remove some of the apps which are not *real apps (i.e., libraries, code snippets, examples)* and others in which the tool we used had problems generating the template (for more information please see Appendix A).

#### 2.4.2 The Final Dataset

We took all the repository URLs appearing on F-Droid and we mined them. We ended up with data coming from 42 apps, summarized in Table 2.2. Due to time constraints, we decided to reduce our data set to focus our analysis on a set of apps selected according the policy illustrated in Table 2.1

Criteria	Apps
Most rated	Diskusage, Search Light and Share My Position
Most installed	Ringdroid, Solitaire for Android and Zxing
Long history	CsipSimple, Mythdroid and Open GPSTracke
Size (in terms of LOC)	Appsorganizer, Replicaisland and Sipdroid
External calls (APIC)	Android VNC Viewer, Open Sudoku and Zirco Browser
High core ratio	Andless, Anstop and App Soundmanager
Random	Alogcat and Mythmote

Table 2.1. Criteria for the selection and corresponding apps.

Name	Rating	Installs	Start rev.	End rev.	Status
Alogcat	4.6	100k - 500k	2	48	•
Andless	4.2	100k - 500k	2	93	•
Android VNC Viewer	4.3	1m - 5m	2	203	N/A
Android WiFi Ace	N/A	N/A	3	14	•
Anstop	N/A	N/A	2	61	•
App Soundmanager	4.5	50k - 100k	1	157	2
Appsorganizer	4.6	1m - 5m	3	191	2
Csipsimple	4.4	100k - 500k	2	1415	•
Cyanogen updater	N/A	N/A	5	657	
Diskusage	4.7	50k - 100k	2	69	2
Droidlife	3.8	5k - 10k	2	23	•
Droidwall	4.6	500k - 1m	186	249	N/A
Flygtider	3.2	5k - 10k	4	23	2
Hotdeath	4.1	10k - 50k	2	38	•
Httpmon	4.1	10k - 50k	7	54	2
Klaxon	4.4	10k - 50k	3	144	•
Mahjonggbuilder	4.0	100k - 500k	2	15	2
Mandelbrot	3.5	10k - 50k	3	152	•
Mathdoku	4.5	100k - 500k	2	73	•
Mythdroid	N/A	N/A	76	640	•
Mythmote	4.6	10k - 50k	2	281	•
Omnidroid	3.9	1k - 5k	453	863	2
Open GPSTracker	4.2	100k - 500k	2	1255	•
Opensudoku	4.6	1m - 5m	15	415	N/A
Quickdroid	4.1	10k - 50k	3	105	2
Replicaisland	4.2	1m - 5m	2	7	2
Ringdroid	4.6	10m - 50m	2	66	2
Scandinavian Keyboard	4.4	100k - 500k	2	17	2
Search Light	4.7	100k - 500k	2	4	•
Share My Position	4.6	10k - 50k	2	76	$\mathbf{c}$
Shelves	3.8	10k - 50k	3	26	2
Sipdroid	4.0	500k - 1m	50	620	2
Solitaire for Android	4.3	10m - 50m	2	30	2
Sparserss	4.2	50k - 100k	2	295	•
Swiftp	4.6	500k - 1m	59	74	2
Tapsoffire	N/A	N/A	7	22	
Tiltmazes	3.1	500k - 1m	11	51	
Trafikinfo.nu	4.0	10k - 50k	2	630	2
Trolly	3.4	5k - 10k	2	40	
Vudroid	3.9	100k - 500k	2	62	•
Zirco Browser	3.8	10k - 50k	65	457	$\mathbf{c}$
Zxing (Barcode scanner)	4.3	50m - 100m	569	2257	•

Table 2.2. List of all apps analyzed in the preliminary study.

For each app Tables 2.2 and 2.3 show the download count and the rating (if available from the Google Play website<sup>8</sup>), the start and end revisions considered in our analysis and the status of the app. Our tool needs to know where is the *source folder* of the project under analysis to obtain consistent data while extracting metrics (more details in Chapter 4). It starts crawling the history of the project from the HEAD revision and goes back until it is possible (*i.e.*, the process stops when the crawler can not find the source folder). We called this point in time *Start revision*. We consider dead ( $\clubsuit$ ) those that were unchanged in the last year. The other symbol ( $\bigcirc$ ) indicates that the app has changed in the last year.

Name	Rating	Installs	Start rev.	End rev.	Status
Alogcat	4.6	100k - 500k	2	48	÷
Andless	4.2	100k - 500k	2	93	•
Android VNC Viewer	4.3	1m - 5m	2	203	N/A
Anstop	N/A	N/A	2	61	•
App Soundmanager	4.5	50k - 100k	1	157	
Appsorganizer	4.6	1m - 5m	3	191	
Csipsimple	4.4	100k - 500k	2	1415	•
Diskusage	4.7	50k - 100k	2	69	
Mythdroid	N/A	N/A	76	640	•
Mythmote	4.6	10k - 50k	2	281	•
Open GPSTracker	4.2	100k - 500k	2	1255	•
Opensudoku	4.6	1m - 5m	15	415	N/A
Replicaisland	4.2	1m - 5m	2	7	
Ringdroid	4.6	10m - 50m	2	66	
Search Light	4.7	100k - 500k	2	4	•
Share My Position	4.6	10k - 50k	2	76	•
Sipdroid	4.0	500k - 1m	50	620	
Solitaire for Android	4.3	10m - 50m	2	30	
Zirco Browser	3.8	10k - 50k	65	457	•
Zxing (Barcode scanner)	4.3	50m - 100m	569	2257	•

Table 2.3. Our App Ecosystem. The dataset used for the detailed investigation.

For each app in Table 2.2, we took several snapshots (*i.e.*, revisions), from less than 10 to more than 2'000. In total we produced more than 10k snapshots that our tool could visualize. Due to time constraints, we decided to reduce our data set to focus our analysis on a set of apps selected according the policy illustrated in Table 2.1. Table 2.3 displays the final dataset: the 20 apps we chosen to analyze. Notice that *qualifying adjectives* are in the context of our apps' dataset: A "*big app*" has 20k LOC, not even comparable to what is considered a "big traditional software system".

<sup>&</sup>lt;sup>8</sup>See https://play.google.com/about/apps/

## **Chapter 3**

## **Related Work**

The aim of our work is understanding how mobile apps are developed and how they differ from traditional software systems. In Section 3.1 we mention some recent studies on mobile applications. Since we are among the first to study mobile applications, we also look at related fields. Areas of software engineering related to our work are Mining Software Repositories (Section 3.2), Software Visualization (Section 3.3), and Software Analytics (Section 3.4).

#### 3.1 Mobile Applications Studies

The first apps were developed only a few years ago. At the time of writing, little research about mobile applications has been performed. In this section we discuss some of the approaches that we found in the literature.

Islam *et al.* studied the global impact of mobile applications on the market, claiming that apps are one of the most rapidly developing areas [IIM10]. They introduced the current state of mobile applications and they discussed the influences of apps on different sectors, including both economical and social aspects. They presented data about smartphones sales, use of broadband connection, and mobile application market and they claimed that all those factors are interconnected and should be studied together. According to them, in the past people were using apps for simple tasks, because of the poor quality and limited functionalities. These days, instead, they argued that software companies produce higher quality products and mobile apps are gradually replacing desktop applications.

Researchers explored also software design aspects of mobile applications. Ruiz *et al.* investigated the reuse in the Android mobile app market along two dimensions: (a) reuse by inheritance, and (b) class reuse [RNAH12]. The authors conducted a case study on thousands of mobile apps, in the Android Market, divided by categories (Cards & Casino, Personalization, Photography, Social and Weather). Among their results, they have shown that, on average, 61% of all classes in each category occur in two or more apps. Hundreds of apps, instead, were reused entirely by another app in the same category.

While the work of Islam *et al.* focuses on marketing aspects, we want to look at apps from a software engineering perspective. To this aim, the work of Ruiz *et al.* is more related to

our approach. Differently from them, we focused on a smaller dataset to conduct an in depth analysis. To support the understanding of the data we used software visualization. To provide a broader context to our research, in the rest of the chapter, we look at the fields related to our work.

#### 3.2 Mining Software Repositories

Software repositories contain a large amount of data that can be used, for example, to understand the evolution of a software system. The Mining Software Repositories (MSR) field [HHM04] analyzes the data available in software configuration management (SCM) systems to discover information about software projects.

Harman *et al.* introduced for the first time the concept of App Store Mining as a form of software repository mining [HJZ12]. The authors used data mining techniques to extract feature information to analyze different aspects of apps. They applied their approach to more than 32 thousand apps available in the Blackberry app store. Their results show that there is a strong correlation between customer rating and the number of downloads. There is no correlation, however, between price and rating, nor between price and downloads.

The study above, at the time of writing, is the only work specific to apps. However, the literature presents a large number of MSR approaches. Among them, we report some representative research topics that can be applied to our study of apps, as part as our future work.

D'Ambros *et al.* [DBL10] empirically studied the relationships between software defects and the presence (and the addition) of a catalog of design flaws over a system's evolution. It is common sense that design flaws are a symptom of low software quality. For this reason, the authors devised a set of design guidelines, such as *"A class should have one responsibility"* and, by inspecting regular snapshots of a software system, checked when those principles were violated (*i.e.*, design flaws). To recognize those flaws, they used detection strategies proposed by Lanza and Marinescu [LM06]. Finally, they analyzed the *deltas* between the addition of new flaws, during the system's evolution, and computed the correlations of both flaw presence and flaw addition. In the end, they showed that no flaw correlates more than others uniformly across systems.

As part as our future work, we plan to investigate on design flaws to assess whether the source code of apps contains the usual code smells, such as duplicated code, long methods or God classes, or if we could define apps-specific smells.

Ball *et al.* conducted the first work on change coupling [BAHS97]. The authors defined two classes as *logically coupled* if they were frequently modified together. They clustered the changes based on modification reports to understand if the changes are distributed, well encapsulated or misplaced.

A possible extension to our analysis is understanding to what extent classes of mobile apps are related to each other (*i.e.*, logically coupled). Investigating on the classes that are frequently modified together may exhibit evolutionary patterns in apps development.

Gîrba *et al.* focused on the expertise of committers. They worked on refining and visualizing code ownership [GKSD05] and introduced the concepts of *line ownership* and *file ownership*. Developer own a line of code (LOC) if they were the last one that committed a change to that line and own a file if they own the largest part of it, in terms of LOC. On top of their approach the authors built visualizations to show development patterns such as monologue, familiarization, or takeover of an author.

In our analysis, we did not consider developer information, but we plan to analyze this factor as part of future work. To this aim, we could apply the approach of Gîrba *et al.* to extract mobile applications development patterns based on the expertise of authors.

Mining software repositories analyzes a significant amount of data. In the literature, among other techniques, researchers used software visualization and software analytics to make sense of this data. In the next two sections we illustrate some work in this areas.

### 3.3 Software Evolution Visualization

Visualization is "the power or process of forming a mental picture or vision of something not actually present to the sight" [SW89]. Software visualization is a specialization of information visualization focusing on software [Lan03b]. Price *et al.* defined it as "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of software" [PBS93]. Software evolution visualization is visualization applied to evolutionary information to support the understanding of software systems.

Lanza introduced *polymetric views*, a lightweight software visualization technique [Lan03b]. A polymetric view is a depiction of software entities and relationships enriched with software metrics. Polymetric views combine static and dynamic information about software systems. For example, a *Method efficiency correlation* view shows all methods using a scatterplot layout; lines of code and number of statements are used as position metrics. The two metrics used are related, thus in the view many methods are aligned along a certain correlation axis.

CodeCrawler [Lan03a] is the tool where Lanza implemented polymetric views for the first time. Wettel and Lanza applied the principles of polymetric views in CodeCity, a tool that uses the city metaphor to visually represent software systems [WL08]. In their visualizations, elements map to a chosen set of software metrics and systems are depicted as interactive, navigable 3D cities. In these cities, buildings represent classes and districts represent packages.

A different approach to visualize the history of software systems, is "The Evolution Matrix" proposed by Lanza [Lan01]. This visualization, displays the history of a software system in a matrix, where rows represent the history of classes, and columns evolutionary snapshots of the system. Each cell in the matrix represents a version of a class where its dimensions are mapped to evolutionary measurements.

Code Flows [TA08] displays the evolution of source code over several versions. The tool models source code using Abstract Syntax Trees (AST) and uses code matching techniques on consecutive ASTs to detect correspondences. The visualization proposed by Telea *et al.* is help-

ful to detect important events such as insertion, deletion, and code drifts.

To have a broader view, Lungu *et al.* considered many software systems at a time, and developed the Small Project Observatory (SPO) [LLGR10]. The authors claimed that many companies must deal with multiple, interdependent, software projects in parallel. To this aim, they defined the concept of software ecosystem as "a collection of software projects which are developed and evolve together in the same environment". SPO is an interactive interface for the exploration and visualization of software ecosystems. The tool provides tables, timelines and graphs to understand the dynamics and interrelationships among projects composing software ecosystems.

In our visualizations we implemented polymetric views, *i.e.*, shapes in our visualizations depict a set of chosen software metrics. We provide also evolutionary visualizations to inspect the evolution of apps, but differently from Code Flows, we do not have means to detect correspondences between snapshots of a software system. The approach of Lungu *et al.* is similar to our work, but differs in the context. Both tools offer highly interactive visualizations, but SPO focuses on a higher level analysis of software ecosystem. Our approach is at a lower level, since we want to analyze source code, and it targets a particular niche of software ecosystems *i.e.*, mobile applications.

#### 3.4 Software Analytics

During the software lifecycle a large number of documents are generated, such as source code, bug reports, SCM meta-data information, and user feedback. Software analytics tries to make sense of these pieces of data extracting software facts used to understand different aspects of software systems, such as how they evolved over time.

Researchers considered different pieces of information when analyzing software systems. For example, Complicity [Neu11] is a tool to analyze software evolution at ecosystem level by means of interactive visualizations. The tools uses data from the webpages of control versioning systems. The tool provides a set of fixed views that can be extended. Neu *et al.* proposed a case study to show how Complicity can help to understand the GNOME ecosystem [NLHD11].

Sarma *et al.* considered additional pieces of information and developed Tesseract [SMWH09], a socio-technical dependency browser that uses cross-linked displays to enable exploration of the relationships between artifacts, developers, bugs, and communications. The tool extracts software facts from SCM files, error logs including communications extracted from their comments, and e-mails but it does not consider source code snapshots. Tesseract uses four juxtaposed displays: (1) the Project activity pane that displays the overall activities in a project for a selected time period; (2) the Files network pane, a graph of dependencies between artifacts; (3) the Developer networks pane displaying relationships among developers and (4) the Issue pane showing defects related information.

Holmes and Begel proposed an approach that considers source code snapshots as a key element for the analysis. The authors developed Deep Intellisense [HB08], a plugin for Microsoft Visual Studio that aggregates information extracted from different sources. In addition to source code, the tool extracts software facts from versioning system meta-data, bug tracking systems, mailing lists, etc. The aim of the tool is to provide developers with intuitions to understand the rationale behind the source code they are looking at, *i.e.*, while working in Visual Studio.

Our approach focuses on three factors: source code metrics, usage of third-party APIs, and generic evolution. Differently from our work, Sarma *et al.* concentrated their research on socio-technical dependencies, while Deep Intellisense is an IDE enhancement that uses different sources to provides developers with a context while programming. Our tool, instead, aims at reverse engineering existing systems to discover pieces of information not available elsewhere. Complicity is similar to our tool but focuses on contributors rather than source code. Moreover, our approach targets mobile applications.

### 3.5 Summing Up

In this chapter we have shown some approaches related to our work describing similarities and differences between our work and previous studies. Since our research lies among the few studies on mobile application we also introduces work in the related fields of software engineering: Mining Software Repositories, Software Visualization, and Software Analytics.

Recently, researchers studied marketing aspects of apps and investigated on software reuse inside the Android Market. Our work mostly focuses on software design aspects of mobile applications rather than marketing. With our work we want to dig into mobile applications to show their peculiarities. Moreover, we want to investigate how apps are developed by inspecting their historical data. To this aim, we have to mine software repositories, the containers of evolutionary data. The MSR process analyze a significant amount of data. To make sense of this large amount of data researchers proposed, among others, software visualization and software analytics approaches. We extract software facts from different sources and provide a catalogue of software visualizations to ease the understanding of these data.

## **Chapter 4**

# SAMOA

This chapter details our tool, *SAMOA*. Section 4.1 gives an overview of the architecture of the tool and Section 4.2 presents the visualizations we devised. The user interface is explained in Section 4.3 and Section 4.4 explains how to interact with the visualizations. The last part (Section 4.5) summarizes the metrics used in the visualizations.

### 4.1 Architecture Overview

*SAMOA* is a web-based tool that allows analysts to visually explore and understand apps. This section depicts the architecture of the tool, summarized in Figure 4.1.

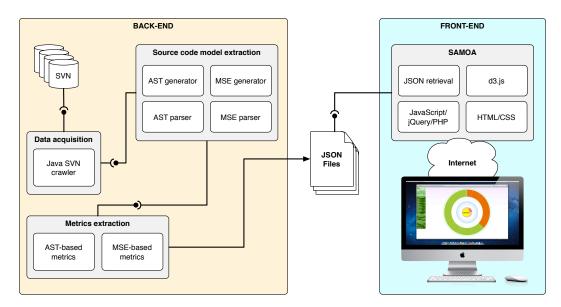


Figure 4.1. Architectural overview of SAMOA

The **back-end** is responsible for gathering the raw data from the repositories, elaborating it and providing an intermediate language between back- and front-end. We chose JavaScript Object Notation (JSON) as exchange language between front- and back-end. In this way the front-end (web application) can easily retrieve and parse them. Four main steps compose this process:

- 1. Obtain the source code of the apps, at different revisions, from their repositories,
- 2. Generate & parse two source code representations (Abstract Syntax Tree (AST) & MSE<sup>1</sup>),
- 3. Extract a set of software metrics from the AST and the MSE file,
- 4. Generate the JSON files, which act as exchange format.

#### Data acquisition

*SAMOA* needs metrics and relations extracted from source code of apps. To this aim, we developed a Java SVN Crawler on top of SVNKit<sup>2</sup>. The crawler obtains the source code of the *HEAD* revision and goes back in history. Our tool needs to know where the source folder of the project is, within the repository. We tried to automate this procedure but some repositories contain more than one app and more than one source folder, thus the entire process became unreliable. To this aim we introduced a constraint: The source folder must be one of the immediate children of the repository path that is fed to *SAMOA*. The crawler continues to work until this constraint is satisfied (*i.e.*, the source code is a children of the root of the repository). After this phase the data is given to the extractors and parsers, described in the next section.

#### Source code model extraction

SAMOA extracts metrics from two sources: the Abstract Syntax Tree (AST) and the MSE file, a generic format, similar to XML, that can describe any model. This file contains complementary pieces of information with respect to an AST, such as details on method invocations, that we used to study the impact on external method calls on Android projects. To generate the AST we used the Eclipse Java Development Tools (JDT) framework that provides facilities to generate and parse AST of Java programs. Concerning MSE we investigated on the two main Moose<sup>3</sup> importers: inFamix<sup>4</sup> and VerveineJ<sup>5</sup>. Moose is an open-source platform for software and data analysis. This work is mantained by several research groups around the world and it is adopted in various industrial projects. Importers are tools that take data from source and produce a model in Moose, indeed an MSE. We evaluated pros and cons of both tool and we tried both of them. The deciding factor in favor of VerveineJ is that inFamix does not model library entities and defaults usages of such entities as unknown. The resulting MSE, produced with inFamix, is an inaccurate model for our analysis (i.e., we cannot model third-party invocations). VerveineJ, instead, models such entities and in addition allows to enumerate class paths explicitly. We used this feature to include the Android SDK in the class path to obtain more precise details on method invocations. Once the two models (AST and MSE) are ready, we feed them to the metrics extractors.

<sup>&</sup>lt;sup>1</sup>See http://www.moosetechnology.org/docs/mse

<sup>&</sup>lt;sup>2</sup>See http://svnkit.com/

<sup>&</sup>lt;sup>3</sup>See http://www.moosetechnology.org/

<sup>&</sup>lt;sup>4</sup>See http://www.intooitus.com/products/infafmix

<sup>&</sup>lt;sup>5</sup>See https://gforge.inria.fr/projects/verveinej/

#### Metrics extraction

The metrics extraction takes place in two phases: Some metrics are extracted from the AST and others from the MSE (more details in Section 4.5). We extract coarse-grained metrics from the AST and fine-grained metrics from the MSE. In case of Android projects there is an additional step consisting of the parsing of the Android manifest, an XML file that presents essential information about the application to the Android system. We used the information contained in the manifest to identify the core of the app (described in Section 2.3.1) and the version number. After the extraction we collect all the metrics and information of the application in JSON files, used as intermediate representation between the back-end and the front-end.

## SAMOA

Figure 4.2 shows *SAMOA* for the first time. This web application represents the front-end of the tool. The exchange format we used to link the back- with the front-end is JSON. A single

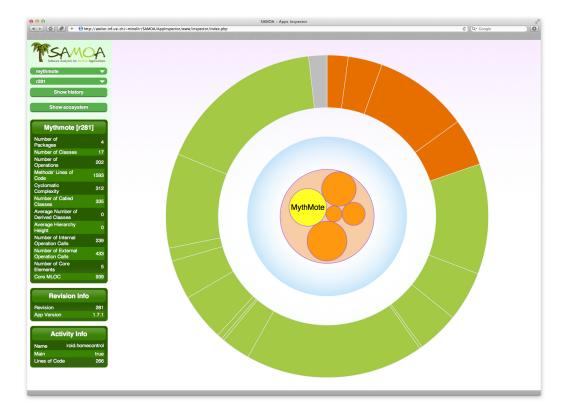


Figure 4.2. A first glance of SAMOA visualizing the MythMote application.

JSON file represents an application at one revision. There are visualizations that requires more than one JSON file at a time (*i.e.*, history visualizations). To this aim the tool, when the user selects an application, retrieves through several asynchronous JavaScript requests (AJAX) all files related to that app (*i.e.*, the entire history). These files are cached for the analysis session

but lost when the user closes the page. We used  $d3.js^6$  to implement the different interactive visualizations. We have 20 apps stored on our server with their entire history. The user can use our tool to understand evolutionary aspects of the apps, such as the LOC distribution throughout the history. If interested she can also compare different applications by visualizing all apps, at the same time, in the ecosystem views. Section 4.2 details the principles behind our visualizations.

## 4.2 Visualizations

Visualizations help interpreting large amount of data. According to the taxonomy of Butler *et al.* [BAB<sup>+</sup>93] our visualizations lie in the category of *explorative visualizations*. This class of views are used when, a priori, the user does not know what to look for. The views we provide help analysts to discover recurring patterns or to spot anomalies in the structure of apps.

We provide visualizations at different granularities: single-app and ecosystem level. Among our views we have *Evolutionary visualizations* that present information coming from the evolution of a software system, complemented with SCM meta-data. The following sections describe the views and explain how the user can interact with them.

## 4.2.1 Single-App Visualizations

*SAMOA* provides two different visualizations of a single system: snapshot- and history-based. The former is used to inspect an app at a given revision (*i.e.*, snapshot) while the latter serves as a mean to understand the distribution of some software metrics during the evolution of the app under analysis. The rest of the section details these two visualizations.

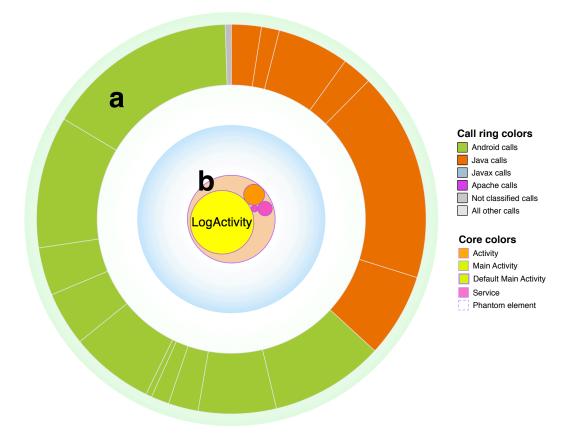
#### **Snapshot View**

Figure 4.3 depicts *SAMOA*'s snapshot view. The visualization is composed of two main entities: the core and the call ring. These visual elements correspond to the concepts described in Section 2.3. We defined the core of the app as the set of classes that are specific to apps' development (*e.g.*, classes that inherit from the mobile platform SDK's base classes). In the case of Android development, we further refined this definition. Core elements are classes defined in the Android Manifest, an XML file that presents essential information about the app to the Android system. The other key entity presented in the visualization is a ring that depicts thirdparty method invocations. The whole span (*i.e.*,  $360^\circ$ ) represents the total number of method calls that refer to external libraries.

The view is useful to explore several dimensions of an app. The call ring (4.3a) shows (1) the number of calls to each of the external libraries (*i.e.*, angle of the slices) and (2) the proportion of internal and external calls. The core (4.3b) presents (1) the core elements (*i.e.*, circles) and the ratio between the total LOC and the coreLOC. The right part of Figure 4.3 shows how to use colors to interpret the visualizations.

<sup>&</sup>lt;sup>6</sup>See http://d3js.org/

The legend in Figure 4.3 distinguishes call's ring and core colors. The formers are used to differentiate the third-party APIs invoked. For example, all calls to *java.io*, *java.lang*, and *java.util* are depicted in red while the calls to *android.app* and *android.view* are painted in green. We assigned colors only to the most recurring libraries among all apps: Android, Java, Javax and Apache. We used two tones of grey to depict both libraries different from the four mentioned above (*i.e.*, lighter grey) and libraries not classified from the extracted meta-model (see Appendix A for more details). Core colors instead identify the type of core element depicted. Core elements can be either Activities or Services. Typically, one Activity in each app is specified as the "main" Activity. The *main Activity* is the one which is presented to the user when launching the app. Shapes with dashed border (*i.e.*, phantom elements), independent from the fill color, are core elements which are defined in the Android manifest but not found by *SAMOA*.



**Figure 4.3.** Single-app visualization and color legend. In this view the main shapes are the calls' ring (a) and the core (b).

Figure 4.4 details the visualization principles. The outer radius of the call ring represents an indication of the current size of the app in terms of both lines of code (LOC) and methods' calls. The green shaded circle represents the max size of the app over the whole history. The example in Figure 4.4 shows a gap between this circle and the outer radius of the calls' ring. This means that in the history there is a snapshot where the app is bigger (*i.e.*, where there is no gap). With this visual clue, when observing different snapshots, one can immediately perceive in which revision the app is bigger.

A similar light blue shaded circle is present to delimit the core. The diameter of this circle is proportional to the total LOC of the application. This circle changes over time accordingly to the size of the app (in terms of LOC) and contains another circle (*i.e.*, the core circle) that depicts the core of the app. The diameter of the core circle (*i.e.*, light red) is proportional to the coreLOC. For example, in Figure 4.4, the core's diameter is (approximately) half the diameter of the core container circle (shaded blue circle). This means that half the LOC of the app compose the core. Core elements, inside the core circle, have sizes proportional to their LOC.

The thickness of the call ring (*i.e.*, the difference between inner and outer radius) is proportional to the number of external calls. The width of the white area between the core container circle and the inner radius of the call ring is proportional to the number of internal calls. This visual clue helps in estimating at a glance the ratio between internal and external method invocations. In Figure 4.4, for example, internal ones are a bit more than half the number of third-party calls.

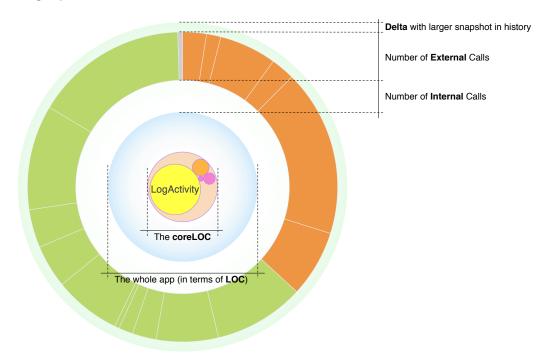


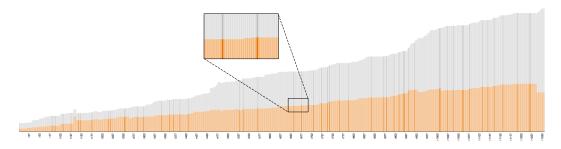
Figure 4.4. The proportions in the single-app view.

This visualization is useful to complement existing sources of information (*e.g.*, SVN logs, source code, etc.) to gather an initial understanding of the structure and behavior of a mobile application. Our visualization provides additional contributions to available pieces of data. The distribution of third-parties method invocations, for example, is an information that is not easily retrievable elsewhere but provides a relevant indication on the external coupling of a software system.

#### **History View**

*SAMOA* provides visualizations that present to the user the entire history of an app. For these views we depict data using two common pattern languages: the histogram and the line chart. To be accurate, we used a particular histogram called *stacked bar chart, i.e.*, a graph used to compare the parts to the whole. In this visualization, each bar represents a snapshot of the application and it is divided into layers according to the kind of data represented. We used line charts to depict the evolution of pieces of data that present no logical layer subdivision (*e.g.*, number of core elements).

We visualized three aspects of apps over their history: (1) the distribution of LOC, distinguishing coreLOC and the rest of LOC, (2) the distribution of the external calls, and (3) the evolution of the number of core elements. Figure 4.5 shows the distribution of LOC of the app *Open-GPS-Tracker* over its 1.2k revisions. Colors used are the same in the legend of Figure 4.3. We used opacity to highlight snapshots where a new version of the app is released. For example, the enlargement in the Figure shows when the authors released version 0.9.18 and 0.9.19.



**Figure 4.5.** A visualization of the distribution of LOC of Open-GPS-Tracker. The magnification in the figure shows the use of opacity to distinguish between successive version of the app.

Figure 4.6 shows the same visualization but with bars grouped instead of stacked. This is useful to see, for example, if coreLOC and the rest of LOC grow together.

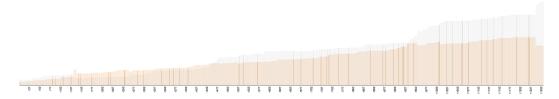


Figure 4.6. The same visualization of Figure 4.5 with grouped bars.

The last evolutionary visualization we proposed is the line chart that depicts the evolution of core elements. In this simple graph to each revision of the application (*i.e.*, x-axis) is associated the number of core elements (*i.e.*, y-axis). Figure 4.7 shows how the number of core elements of the *Zirco-Browser* app vary over time.

25

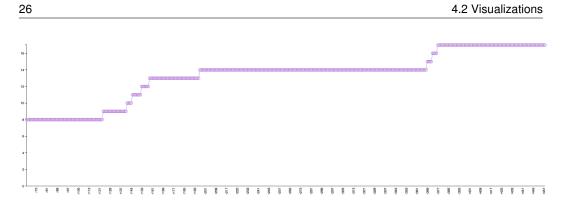


Figure 4.7. A visualization of the distribution of the number of core elements of Zirco-Browser

In this section we have not shown the stacked bar chart that depicts the distribution of the external calls. In addition to single-app views, we provide visualizations at a coarser granularity (*i.e.*, ecosystem). Among them there is the bar chart presenting method invocations. To this aim, this visualization is detailed in the next section together with other ecosystem views, useful to have a broader prospective.

## 4.2.2 Ecosystem Visualizations

The visualizations introduced in the previous section consider only a single app at a time. To have a wider view we can consider "a collection of software projects, which are developed and co-evolve together in the same environment". This is how Lungu defines the concept of software ecosystem [Lun09]. We argue that a set of apps can be seen as an ecosystem. To this aim, we developed two visualizations at ecosystem level: the grid view and the stacked bar chart view.

## **Grid View**

The grid view is a simplification of the *snapshot view* applied to multiple apps at the same time. Figure 4.8 shows 10 apps in a grid view. Each shape in the visualization represents an app. The diameter of each shape is proportional to the number of total LOC while the diameter of the core circle (*i.e.*, yellow) corresponds to the value of the coreLOC metric.

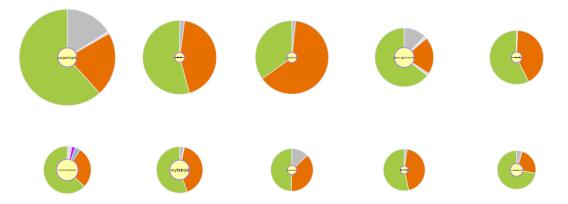


Figure 4.8. A grid view of an App Ecosystem. Apps are ordered by size (in terms of LOC)

The ring surrounding the yellow circle represents the external calls. The whole span of the ring  $(360^\circ)$  portrays the total number of calls. The width of each of the slices is proportional to the number of calls to the library identified by its color (*i.e.*, the color mapping is shown in Figure 4.3). Its thickness is not mapped to any metric related to methods' calls. Apps are grid-aligned and ordered according to their size (in terms of LOC).

The same view is available with a different partitioning function for the calls' ring. Since elements in this visualization are small, the user can rearrange the calls' rings with an equal breakdown among slices. For example, if an app uses four external libraries, each sector of the calls' ring will have an angle of 90°. Figure 4.9 shows the first two rows of the grid of Figure 4.8 with the new partition function applied.

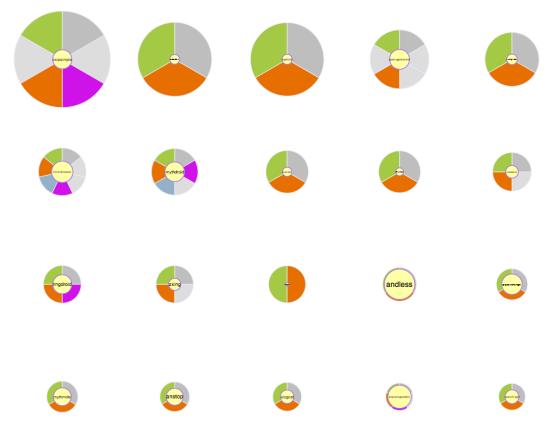


Figure 4.9. Another grid view of an App Ecosystem with unitary partition of methods' calls.

#### **Stacked Bar Chart View**

We used the same principles of the *history visualization* (described in Section 4.2.1) to depict software ecosystems. Each bar represents a snapshot of a different app, instead of different snapshots of the same app, as in the *history view*. Snapshots refer to the *HEAD* revision of each application. Figure 4.10 shows the calls' distribution of the ecosystem, composed of 20 apps.

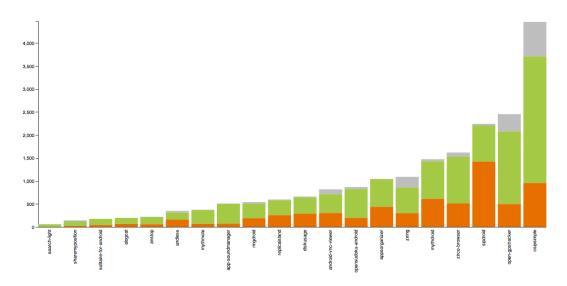


Figure 4.10. External calls distribution of the app ecosystem.

This visualization can present different kinds of data. In addition to the distribution of method invocations we provided also the distribution of number of LOC and number of commits (*i.e.*, revisions). For the number of revisions there is a single layer, so the graph became a classic histogram. Figure 4.11 shows the distribution of the number of commits.

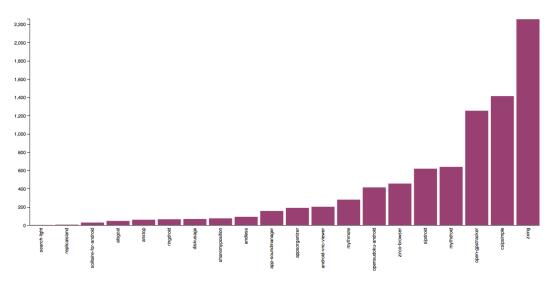


Figure 4.11. Number of commits among the app ecosystem.

The aim of our bar charts is to visually compare the proportions, and not the real values. For example, the visualization above shows that *open GPS tracker* has twice the number of commits of *mythdroid*. However, for the sake of completeness, *SAMOA* depicts the x-axis with the scale and provide means to assess the exact values of all fragments composing the bars, by means of simple interactions, described in Section 4.4.

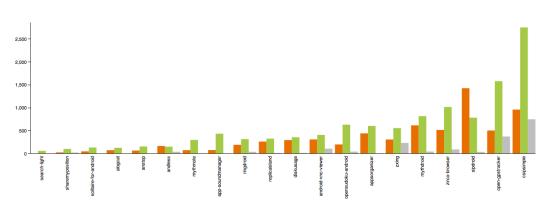


Figure 4.12. The same visualization of Figure 4.10 with grouped bars.

This visualization can be interactively resorted, according to one of the layers, and bars can be grouped. By default bar charts are sorted by the sum of the values of each layers (*i.e.*, the bars of the LOC distribution are sorted, in ascending order, by the total app's LOC). Figure 4.12 shows a chart of the calls' distribution with grouped bars while Figure 4.13 depicts the LOC distribution, sorted by size of the core (*i.e.*, coreLOC).

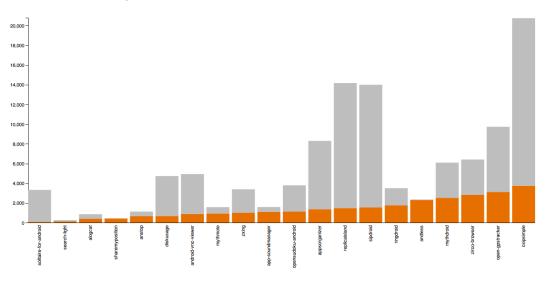


Figure 4.13. Distribution of LOC sorted in ascending order according to coreLOC.

Ecosystem visualizations provide overviews of a large amount of data. In our context, they are useful to spot the first peculiarities of apps and decide which set of apps are worth being investigated. We used these views to shrink our dataset, as explained in Section 2.4.2, to choose the candidate apps for our detailed analysis.

## 4.2.3 Discussion

We provide two classes of single-app visualizations: snapshot- and history-based. The former reveals more details with respect to historical views. On the other hand, visualizing the whole

evolution of a software system highlights different facets that are hard (or impossible) to assess from our snapshot-based visualization.

We argue that history-based visualizations are useful to decide on which snapshots to concentrate the analysis. The common use case of our tool is, after selecting a single application, to browse the distributions of LOC and calls to understand where it is worth to do a deeper investigation. For this reason, during the analysis, the user must give equal importance to both categories of views. The pieces of data presented in one type of visualization must be complemented by the view at the other granularity.

We consider ecosystem visualizations equally important. The domain of our investigation (*i.e.*, mobile apps) is not well known. Views at ecosystem granularity allow analysts to have a broader perspective when starting their analysis tasks. These visualizations help to decide which apps are worth investigation, according to either total size, size of the core, or external method invocations.

## 4.3 User Interface

The user interface of *SAMOA* is minimal to avoid distractions from the analysis task. Elements in the visualizations, instead, are depicted with bright colors to attract users' focus.

The surface is divided into 3 main parts, as shown in Figure 4.14:

- The tools panel (1) in the top left corner,
- The inspection panel, composed of:
  - The metrics table (2),
  - The revision info panel (3),
  - The sliding panel (4), which shows additional information of the shape currently hovered in the visualizations.
- The visualization space (5).

#### **Tools Panel**

It gives the possibility to (a) change the app (and revision of the app) under analysis, (b) toggle history-based visualizations of the current app, and (c) switch to ecosystem visualizations. This panel changes according to the context. When either the ecosystem or the history is visualized this panel (d) provides options to change layout and rearrange the items in the visualizations.

#### **Inspection Panel**

This panel displays all the available metrics (a). Depending on the context, single-app or ecosystem, it shows metrics of an application or of the entire ecosystem. In the *snapshot view* — described in Section 4.2.1 — it works also as zoom controller and re-initialization for the visualization (b). By scrolling (*mouse wheel*) on the panel the zoom level is changed and by right-clicking the visualization is reset to the original position.



Figure 4.14. Single-app view of SAMOA visualizing the Alogcat application at revision 48

#### **Visualization Space**

The rest of the page is available for visualizations. All the views automatically fit themselves in the browser's window and the user can interact with all of them, as described in Section 4.2.

## 4.4 Interacting with the Visualizations

*SAMOA*'s views are interactive. The user can perform a series of actions to update the visualizations or obtain context-sensitive information. We present the basic navigation of the tool in Section 4.4.1. Section 4.4.2 shows how to interact with the *snapshot view* and interactions with the *grid view* are explained in Section 4.4.3. Finally, Section 4.4.4 presents interactions with the *bar charts view*.

## 4.4.1 Basic Navigation Interactions

When the user visits *SAMOA*'s homepage, the tool shows a page containing some information about its features and presents to the user the first two possible actions that the she can do. At the beginning the user can pick an app from the drop-down menu and visualize it or ask *SAMOA* to visualize the whole app ecosystem. If the user chooses to visualize a single app (*e.g.*,

*AndLess*), *SAMOA* displays the page depicted in Figure 4.15. The **Tools Panel** (see Section 4.3) automatically updates and provides the user with new possibilities. He/she can (1) change the app being analyzed, (2) move to a different snapshot of the same app, (3) show the history of the current app, and (4) show the app ecosystem.

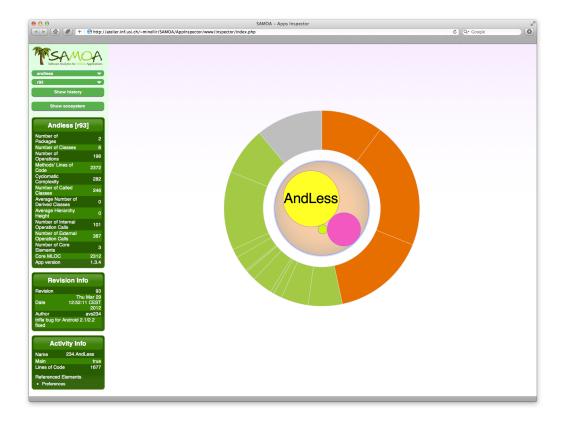
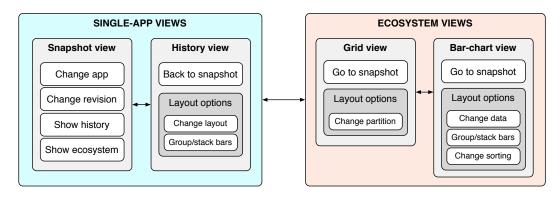


Figure 4.15. SAMOA displaying a snapshot view of AndLess

For the first two interactions, SAMOA provides keyboard-shortcuts to ease the navigation:

- $[\rightarrow]$  visualizes the next app in the list,
- [←] visualizes the previous app in the list,
- [ 1] goes to the previous snapshot of the current app (if any),
- $[\downarrow]$  goes to the next snapshot of the current app (if any).

The **Tools Panel** is context-sensitive. Depending on the current page it offers different navigation options. For example, from the *history view* one can only change the layout of the view or go back to the snapshot of the app. Figure 4.16 illustrates the available options offered by the Tools Panel, depending on the context.



**Figure 4.16.** Navigation possibilities and possible paths offered by the Tools Panel. Light gray boxes represent the context while inner boxes options, possibly nested. Connections between boxes show the paths one can follow during the analysis.

## 4.4.2 Snapshot View Interactions

This visualization, detailed in Section 4.2.1, is the most interactive. The user can zoom and pan the shapes in the view. The **Inspection Panel** (see Section 4.3) controls the zoom level: By scrolling the mouse wheel over this panel the shapes in the view are zoomed. To pan the visualization the user has to drag & drop the visualization with his mouse. In case the view is outside the browser's window, the user can right-click on the Inspection Panel to reset position an zoom of the view to the original settings.

The *mouseover* gesture is used to reveal additional information. If the user hovers on a shape in the visualization the **Sliding Panel** appears (if hidden) and shows extra details of the shape. For example, hovering on a core element reveals its LOC, the type of the element (*i.e.*, Activity or Service) and the referenced elements (if any).

Core elements, most of the time, have their original source code associated to them. *SAMOA* allows the user to browse source code of core elements. To visualize it, the user has to click on the core element he is interested in. If source code information is available, an overlay containing the *pretty-printed* file appears. Figure 4.17 displays the source code viewer. To close it, the user has to click on it.

## 4.4.3 Grid View Interactions

The grid view allows three types of interactions. Using the drop-down option from the **Tools Panel** the user can change the partitioning function for the calls' ring. *SAMOA* depicts sectors either with a span proportional to the number of method invocations or with unary weight (*i.e.*, if there are 6 library referenced, each sector would have a span of  $360^{\circ}/6 = 60^{\circ}$ ). This is useful to spot all libraries (*i.e.*, colors) that each application uses, independently from the number of invocations.

*Mouseover*, consistently with all other views, helps to unveil extra information. This is achieved by showing and updating the **Sliding Panel** with the most recent content. In this



**Figure 4.17.** *SAMOA*'s source code viewer. In this prototype of *SAMOA* the source code refers always to the last revision. This implies that (1) if a class is not present in the last revision, *SAMOA* cannot visualize its source code and (2) if the user is visualizing an earlier snapshot and clicks on a core element, *SAMOA* visualizes the file at the last revision.

visualization additional information includes: number of revisions, LOC, coreLOC, internal, and external calls. The last interaction with this view concerns the change of scope. From this view the user can jump to the *snapshot view* by clicking on the corresponding shape.

## 4.4.4 Bar Chart Interactions

Our tool provides bar chart visualizations for both single-app (*i.e.*, history) and ecosystem views. There are some differences in the interactions. In particular, the *history view* cannot be altered much. Bars represents snapshots and are sorted, in ascending order, by revision number. The two interactions that the user can perform are (1) group and sort the bars (using the buttons), and (2) hover on a bar to discover extra information, and (3) click on a bar to visualize the snapshot of the app at that revision.

The *ecosystem bar chart*, in which bars represent snapshots of different apps, in addition to the interactions described above, allows to re-sort the data. The tools panel for this view, allows to (1) hide the ecosystem, (2) change the data origin of the chart, (3) re-sort the bars

according to one of the layers, and (4) group (or stack) the bars, looking at the data from a different perspective.

Available options depend on the data. For example, the revision bar chart cannot be neither re-sorted nor its bars can be grouped, since it only has one layer. The *mouseover* and the *click* behaviors are consistent with what explained for the single-app bar chart: Mouseover reveals additional information and click shows the snapshot of the clicked app.

## 4.4.5 Line chart Interactions

The interactions with the line chart visualization are limited to *click* and *mouseover*. Consistently with other visualizations, clicking on a data point (*i.e.*, that represent a revision) shows the snapshot view of the selected version and the *mouseover* gesture unveils additional information about the entity.

## 4.5 Metrics

There are different types of software metrics that measure multiple aspects of software systems. Lorenz & Kidd [LK94], defines as *Design Metrics* the metrics used to determine size, quality, and complexity of software systems. The aim is to evaluate the design level of a snapshot of a system. The second category, in their taxonomy, is *Project Metrics*. Those metrics are at a higher level of abstraction and focus on the dynamics of a project. These help estimating what it takes to reach a specific point in the development cycle or how to recognize that one is already there.

Our visualizations are enriched with a set of software metrics. Since the key factors of our analysis are core elements and method invocations, we collected some measures on size of core elements (*i.e.*, lines of code) and we characterized the distribution of methods invocations. To this aim, for each app, *SAMOA* includes the following metrics:

- **INTC** *Number of Internal Calls. i.e.*, the number of invocations that implements internal behaviors (*i.e.*, calls to another user-defined method).
- **EXTC** *Number of External Calls. i.e.*, the number of invocations that refers to third-parties libraries. (*i.e.*, calls to the JDK, to Apache, to the Android SDK, etc).
- NOCE *Number of Core Elements. i.e.*, the number of classes we defined as core of the apps. (*i.e.*, for Android the classes appearing in the Manifest).
- **CoreLOC** *Core's Lines of Code. i.e.*, the lines of code composing the core. The sum of the lines of code (*i.e.*, LOC) of core elements.

We consider the history as a key factor for our analysis. For this reason, we enhanced our set of metrics with two additional pieces of information aimed at characterizing the evolution of mobile applications:

- **App Version** *i.e.*, the release version of the app. (*i.e.*, for Android app it is defined in the Manifest).
- **Revision information** *i.e.*, the information obtained from SCM meta-data. (*e.g.*, revision number, date, author and log message).

In addition to the ad-hoc metrics used in our analysis, *SAMOA* includes a complementary set of software metrics that belong to the *Design Metrics* category, according to Lorenz & Kidd. This additional measures were inspired by the Overview Pyramid of Lanza & Marinescu [LM06], a means to characterize software systems, based on software metrics:

- NOP Number of Packages. i.e., the number of Java packages within the project.
- NOC Number of Classes. i.e., the number of user-defined classes.
- NOM Number of Methods. i.e., the number of user-defined methods.
- LOC *Lines of Code. i.e.*, the number of lines of code of user-defined methods. This measure counts only non-empty LOC within methods' bodies.
- **CYCLO** *Cyclomatic Number. i.e.*, the sum of McCabe's cyclomatic number [McC76] for all methods.
- **CALLS** *Number of Method Calls. i.e.*, the total number of distinct invocations in the project. It is the sum of the distinct invocations for each user-defined method.
- **FANOUT** *Number of Called Classes. i.e.*, the sum of the FANOUT [LK94] metric for all user-defined methods.
- ANDC Average Number of Derived Classes. i.e., the average number of direct subclasses of user-defined classes. This metric does not count Java interfaces. It is a measure of the breath of inheritance.
- AHH Average Hierarchy Height. i.e., the average of the Height of the Inheritance Tree (HIT) among the system's root classes. A class is a root if it is not an interface and if it is not derived from another user-defined class within the system. It is a measure of the depth of the inheritance.

SAMOA shows software metrics in its **Inspection Panel**. Metrics displayed depend on the context. *i.e.*, if the user is analyzing a system's snapshot the tool shows metrics referred to the single project. Otherwise, if the ecosystem is displayed, the Inspection Panel contains coarser-grained measures of the whole ecosystem. To this aim, we devised a set of ecosystem metrics aggregatint single-app metrics. The Inspection Panel of the apps ecosystem contains the following measures:

- Number of apps i.e., the number of apps composing the ecosystem.
- Min/Max/Total LOC *i.e.*, the minimum, maximum, and cumulative values for the LOC metrics among the apps in the ecosystem.
- **Min/Max Number of Commits** *i.e.*, the minimum and maximum number of commits among the apps in the ecosystem.
- Average Calls Ratio *i.e.*, the average ratio between INTC and APIC among all the apps in the ecosystem.
- Average Core Ratio *i.e.*, the average ratio between CoreLOC and LOC among all the apps in the ecosystem.

We used different sources to extract metrics. For each app we generated and parsed the Abstract Syntax Tree (AST) and the MSE file, a generic file like XML, that can describe any model. In particular, from the **AST** we extracted: *NOP, NOC, NOM, and LOC*. We used the **MSE** to obtain values for *CYCLO, FANOUT, ANDC, AHH, INTC, and APIC*. Finally, *Core elements* and *app's version numbers* were obtained by parsing the **Android manifest**.

## 4.6 Summing Up

In this chapter we described *SAMOA*, our mobile application analytics platform. We first illustrated the architecture of the tool, explaining how the back- and the front-end work together and their limitations. Sections 4.2 detailed our views and explained the visualization principles. We also discussed the importance of both categories of views *i.e.*, single-app and ecosystem. We further detailed single-app visualizations, distinguishing between snapshot- and history-based views. We argued that, during the analysis, the user must take advantage of all the analysis options provided by *SAMOA* to inspect apps from different, but complementary, perspectives. The chapter continued with a description of the user interactions allowed by our tool. The final section served as an explanatory index of the software metrics we named in the course of this chapter. After having detailed *SAMOA*, in the next chapter, we present the catalogue of characteristics of mobile applications that we collected during our analysis.

## **Chapter 5**

# A Catalogue of Characteristics of Mobile Applications

In this chapter we present a *catalogue* of peculiarities of mobile applications. We describe characteristics of mobile applications which have emerged from the use of our visualizations (*i.e.*, *visualization-driven*). We divided the catalogue in two categories: snapshot- and history-based, according to the type of view used (See Section 4.2.1 for details on the visualizations). The catalogue is complemented by examples, possible symptoms and scenarios we faced using *SAMOA* to analyze the app ecosystem (See Section 2.4).

## 5.1 Snapshot-based

We identified some properties of mobile application that can be inferred from our *snapshot view*. In this section we describe how to use our visualization to discover software facts that are typical to apps' development.

This part of the catalogue illustrates the following characteristics of apps:

- Dominance of external calls (p. 40)
- Dominance of internal calls (p. 41)
- God-core classes (p. 42)
- High core ratio (p. 43)
- Low core ratio (p. 44)
- Multiple main activities (p. 46)

## 5.1.1 Dominance of External Calls

**Scope**: Method invocations.

**Description**: External calls are more than 75% of the total number of method invocations. **Symptoms**: The inner radius of the call ring, shown in Figure 4.3b, is close to the radius of the core container circle (*i.e.*, shaded blue circle). External calls are represented by the thickness of the call ring.

**Scenario**: We show the Share My Position app in Figure 5.1. In the example, the app has 157 external invocations and only 48 calls that implement internal behavior.

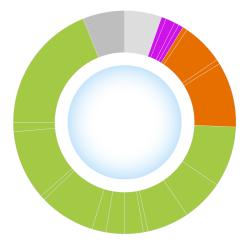


Figure 5.1. Dominance of External Calls in Share My Position application. External calls are represented by the thickness of the call ring.

Apps rely on third-party API. We visually inspected many apps, using the visualizations of our tool, and most of them have a strict dominance of external calls. Among apps the most popular libraries used are the Android SDK and the Java Development Kit (*i.e.*, respectively depicted in light green and red in our visualizations). The other third-party libraries that are commonly referenced are Apache and Javax.

In our visualizations we considered method invocations as an important factor to observe. The snapshot view provides means to visually determine how such calls are distributed and what is their impact on the total number of invocations. However, we have not exhaustively examined the behaviors of external calls in each application. In this respect, it remains to be investigated — if and how — third-party invocations can be used to classify apps and if those calls can help us to deduce mobile apps development patterns. For example, as part as our future work, we plan to study apps that contain invocations to the same set of libraries. The aim is understanding if calls to the same libraries imply other similarities between apps, such as structural or behavioral properties.

## 5.1.2 Dominance of Internal Calls

**Scope**: Method invocations.

Description: Internal calls are more than 75% of the total number of method invocations.

**Symptoms**: The call ring is thin. The white area between the inner radius of the call ring and the core container circle is used to depict internal calls. Having many internal calls produces a thin call ring.

**Scenario**: Figure 5.2 shows an app with dominance of internal calls: Replica Island. The app counts more than 4.3k method invocations, 708 of which are invocations to third-party libraries. The remaining calls (*i.e.*, more than 3.5k) implement internal behavior.

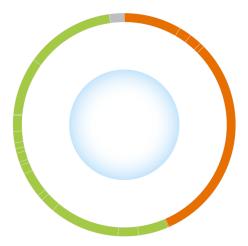


Figure 5.2. Dominance of internal Calls in Replica Island application. Internal calls are represented by the thickness of the white area between the inner radius of the call ring and the core container circle.

In the Replica Island example, the app counts 141 classes, 10 of which are core elements. This lets us assume that the app has a small part that is specific to Android (*i.e.*, the core) and lots of functionalities implemented elsewhere. Further research can be done to understand, for example, if apps having this shape are porting of apps originally developed to run as standalone Java applications.

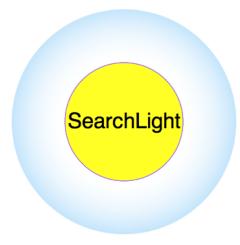
## 5.1.3 God Core Classes

**Scope**: Core of the application.

Description: The core is composed by a single class (i.e., God-class [Rie96]).

Symptoms: The core circle is filled with a single core elements.

**Scenario**: Figure 5.3 depicts the SearchLight application. The core is composed by a single class of 143 LOC.



**Figure 5.3.** God-class smell in the SearchLight application. The core is composed by a single class, SeachLight, the Main Activity. Probably our definition of God-class here is excessive. Some apps are so simple that one class — with its single responsibility — does the job.

Sometimes apps are so simple that does not require much programming. For example, in our scenario the class that we called *god-class* is not a *real* god-class. SearchLight is a class with few LOC, and probably is not overloaded of functionalities at all. The SearchLight application just switches on the camera's LED. The main activity does just that. Further investigation is required to see if apps present traditional god-class smell.

## 5.1.4 High Core Ratio

Scope: Core of the application.

Description: CoreLOC are more than 75% of the total LOC of the system.

**Symptoms**: Small gap (or no gap) between the core circle (*i.e.*, light red circle) and the core container (*i.e.*, shaded blue circle)

**Scenario**: Figure 5.4 shows the AndLess application at revision 93 using our snapshot view. The application has 2372 lines of code, 60 of which are non-coreLOC. The remaining LOC compose the core of the application (*i.e.*, Android Activities & Services).

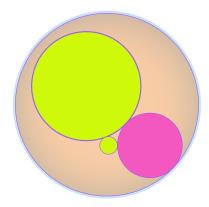


Figure 5.4. High core ratio symptoms in the AndLess application.

Applications following this pattern have the whole logic implemented within core elements. This may results in violations of design guidelines (*i.e.*, design flaws), such as *single responsibility principle*. In common sense, design flaws are a symptom of low software quality. AndLess, for example, is a music player for Android. Its main activity (*i.e.*, the big yellow circle in Figure 5.4) is a class that in addition to drawing the UI is responsible for start & stop the music, recursively traverse the file system to find the music, parse playlists and handle CUE files, etc. We claim that this class is probably overloaded, since violates the *single responsibility principle*, and worth investigation.

## 5.1.5 Low Core Ratio

Scope: Core of the application.

Description: Non-CoreLOC are more than 75% of the total LOC of the system.

**Symptoms**: Very large gap between the core circle (*i.e.*, light red circle) and the core container (*i.e.*, shaded blue circle). The core circle is much smaller than the core container.

**Scenario** #1: We show a snapshot (r203) of the Android VNC Viewer application in Figure 5.5. Its core is composed by 892 LOC out of about 5k.

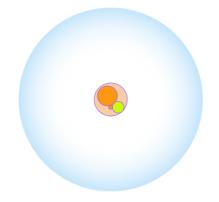


Figure 5.5. Low core ratio symptoms in the Android VNC Viewer application.

Applications presenting this blueprint are probably worth investigation. For example, the Android VNC Viewer is a Virtual Network Computing (*i.e.*, VNC) application that lets the user remotely access his/her PC using its Android phone. We discovered that the core ratio was low because of the large amount of logic of this application. The authors implemented a custom handling of gestures on the multi-touch Android screen as well as all the remote communication mechanisms to serve VNC functionalities. This features were not implemented inside Activities and Services (*i.e.*, the core), but core elements use those helper classes. In other apps, for example, we discovered that the imbalance between core and non-core classes was due either to a large amount of third-party source code imported inside apps or auto-generated code produced by some external libraries.

**Scenario** #2: The Sipdroid app presents the same peculiarity: Non-core lines of code are significantly more than coreLOC. Figure 5.6 depicts a snapshot (r620) of this application. At this revision, the app counts more than 14k LOC, 1'500 of which compose the core of the app.

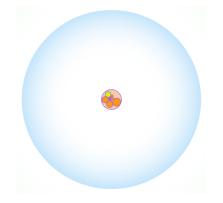


Figure 5.6. Low core ratio symptoms in the Sipdroid application.

We investigated on this app and we discovered that it uses JSTUN<sup>1</sup> *i.e.*, a Java-based library for Simple Traversal of User Datagram Protocol through Network Address Translation. Sipdroid is a Voice Over IP client and must deal with firewalls and NATs. To this aim, it uses JSTUN, a means to discover the existence and type of firewalls and NATs between an application and the public internet. The particular fact is that Sipdroid includes all the source files of the JSTUN library instead of the compiled version (*i.e.*, JARs). For this reason there is an important gap between core and non-core LOC. It remains to be investigated the rationale to include inside apps the entire source code of third-party libraries.

<sup>1</sup>See http://jstun.javawi.de/

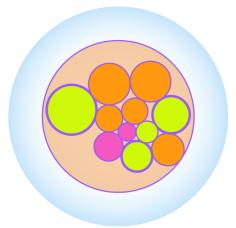
## 5.1.6 Multiple Main Activities

Scope: Core of the application.

**Description**: In Android apps must have a Main Activity. From the Android Developer Guide "an application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time". During our analysis we discovered that many apps have more than one Main Activity.

**Symptoms**: The core of the application contains more than one core element painted in yellow (*i.e.*, the color we used in the visualizations to highlight Main Activities).

**Scenario**: Figure 5.7 depicts the App SoundManager app. The core has 8 Activities (*i.e.*, orange and yellow) and 2 Services (*i.e.*, purple). Four of the activities are labeled as "main" and three of them are "default" main activities.



**Figure 5.7.** Multiple main activities in the App SoundManager application. Main activities are depicted in yellow and *"default"* main activities have a thicker stroke.

Multiple main Activities designate several entry points for the app. To specify which Activity the OS (*i.e.*, Android) has to run when the app is launched for the first time, developers have to specify in the manifest a *default* Activity (*i.e.*, android.intent.category.DEFAULT). The AndLess application (see Figure 5.4), has two main Activities, and one of them is correctly identified as "default" (*i.e.*, thicker stroke in the view), thus the Android system knows which activity to run first. Figure 5.7 — however — shows that some apps (*e.g.*, App SoundManager) have multiple "default" main activities. It remains to be investigated why some apps have diverse entry points and how Android behaves when more than one "default" main activity are specified.

## 5.2 History-based

Some characteristics of apps can be inferred by looking at the *history visualization*. This view depicts the distribution of some measures (*e.g.*, external calls and LOC) during the evolution of the application.

This part of the catalogue details the following characteristics of apps:

- Core drop (p. 48)
- Delayed use of versioning systems (p. 49)
- Flat intervals in history (p. 50)
- Gradual increase of core elements (p. 51)
- High correlation between LOC and third-party calls (p. 52)
- Out-of-sync manifest file (p. 54)
- Snap decrease of core elements (p. 55)
- Snap increase of core elements (p. 57)
- Stepwise growth in history (p. 58)
- Stepwise increase of core elements (p. 59)

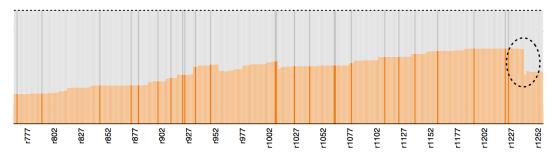
## 5.2.1 Core Drop

Scope: Distribution of the values of LOC.

**Description**: The distribution of LOC shows a drop of core LOC while the total LOC (*i.e.*, the total height of the bars) remains almost unchanged.

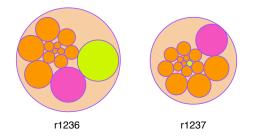
**Symptoms**: The bar chart graph of the history presents a decreasing jump in the core LOC (*i.e.*, the height of the red piece of the bar).

**Scenario**: Figure 5.8 depicts part of the history of the Open GPS Tracker app. This system presents a core drop between revisions 1236 and 1237 *i.e.*, the coreLOC value decreases by 700.



**Figure 5.8.** Core drop in the Open GPS Tracker application. The significant jump in the distribution of core LOC is highlighted. Notice that the figure is cropped, but the cumulative height of the two adjacent bars (*i.e.*, revisions 1236 and 1237) remained unchanged.

We investigated on this issue using our snapshot visualization. Figure 5.9 shows the two core snapshots of the application at revisions 1236 and 1237. One must note that the number of core classes increases (*i.e.*, from 13 to 15) even though the value of coreLOC decreased. Our assumption is that some logic that overloads some core elements (*i.e.*, the big yellow element) was moved outside the core. The log message confirms this hypothesis, saying: *"Split OSM and Google in different Activities with a common helper"*. The comment motivates both the addition of two new core elements (*i.e.*, split in different Activities) and the reduction of the core. It remains to be investigated if this event happens elsewhere and if it is always caused by refactoring activities.



**Figure 5.9.** A comparison of the two core snapshots presenting the core drop in the Open GPS Tracker application. On the left revision 1236 and on the right revision 1237. The total LOC of the apps remained the same while from the core were removed 700 lines of code.

## 5.2.2 Delayed Use of Versioning Systems

Scope: Distribution of the values of LOC.

**Description**: Software repositories tracks and provides control over changes happening to the versioned data (*e.g.*, source code, documentation, etc). Since software is subject to continuous growth from the beginning of its evolution [LPR<sup>+</sup>97], it is common sense to use versioning systems in early phases of a project. Apps' developers, sometimes, violate this habit *i.e.*, some repositories, at the first revision, already contain the full app.

**Symptoms**: The first bars, in the LOC distribution graph, have (approximatively) the same height of bars at subsequent revisions.

**Scenario**: Figure 5.10 shows the history of Solitaire for Android. At the beginning of the history the app counts 2.5 kLOC. Our assumption is that the app had a previous evolution, that was not versioned, or versioned elsewhere. The commit comment proves we were right by saying: *"Initial add, corresponds to market version 1.8"*.

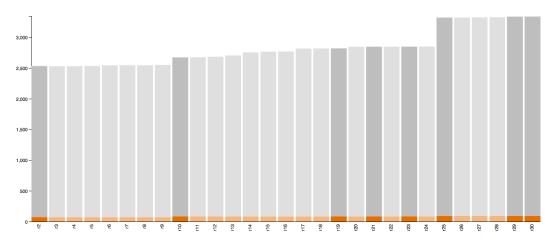


Figure 5.10. Delayed use of SVN in Solitaire for Android application. First revision has 2.5 kLOC.

This unusual practice is repeated in other apps. It can be argued that these apps were not open source until a certain moment in time. Then developers decided to publish their source code using public project hosting services, such as Google Code.

## 5.2.3 Flat Intervals in History

**Scope**: Distribution of the values of LOC (and external calls). **Description**: The distribution of calls and LOC during the history shows some flat intervals.

**Symptoms**: Some adjacent bars in the history bar chart have the same height. **Scenario**: The Share My Position app presents different flat intervals in the LOC distribution, as shown in Figure 5.11.

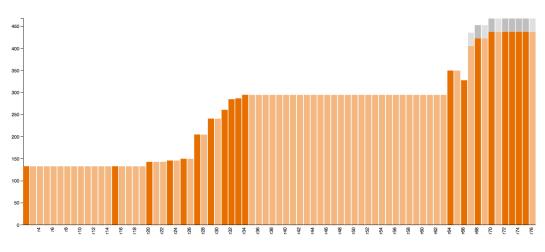


Figure 5.11. Flat intervals in the LOC distribution of Share My Position.

More than one app presents this shape. We investigated on the causes by looking at auxiliary pieces of information, such as SVN logs. In some cases, a single repository contains multiple applications, or versions of the app for different platforms. In the example above, the developers are implementing a version of the app for a different platform (*i.e.*, iOS). The other most frequent reason for such intervals, is the creation or update of the app's documentation (*e.g.*, Wiki page), not captured by our tool.

50

## 5.2.4 Gradual Increase of Core Elements

Scope: Distribution of the number of core elements.

Description: Over the history the number of core elements increases gradually.

**Symptoms**: The line chart showing the distribution of the number of core elements follows a gradual increasing trend.

**Scenario**: Figure 5.12 depicts the evolution of the number of core elements of the Zirco Browser app. In this system the number of core elements grows regularly as the app evolves. Our visualization shows this behavior by means of a line that assumes increasing values from the beginning of the graph through its end. In particular the Zirco Browser application has 8 core elements at its first available revision (r65) and reaches 17 core elements at revision 457.

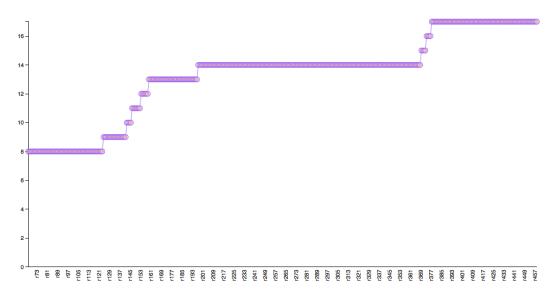


Figure 5.12. The distribution of the number of core elements of the Zirco Browser application.

This behavior is typical to most of the apps we analyzed. This pattern means that the app started with a limited subset of functionalities and, during its evolution, it is enhanced with the addition of new Activities and Services (*i.e.*, core elements). Core elements represent focused tasks that the user can do in the application, thus their addition is aimed at enhancing the functionalities of the app.

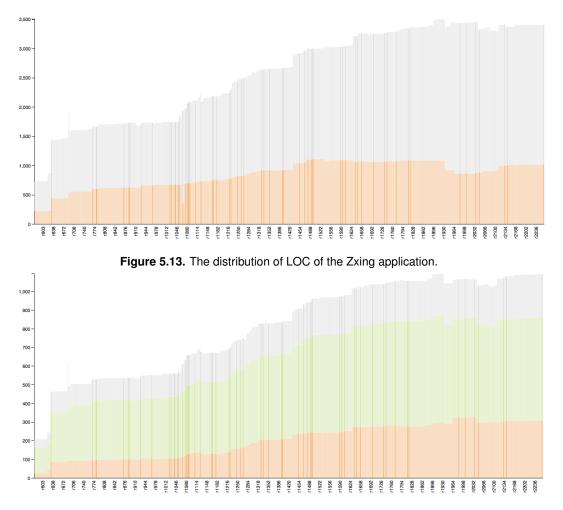
## 5.2.5 High Correlation Between LOC and Third-Party Calls

Scope: Distribution of the values of LOC (and external calls).

**Description**: In Section 2.2, we shown that there is a connection between LOC and external method invocations. In particular, the correlation coefficient between these two metrics, among the whole dataset, is high (*i.e.*, 0.84).

**Symptoms**: This relationship between LOC and third-party method invocations is visible from our history visualizations. Among the pieces of data that these views can show, there are the distributions of LOC and external method calls. In most apps in our dataset those two distributions have almost identical shapes.

**Scenario**: Figure 5.13 and 5.14 respectively show the distribution of LOC and external methods invocations of the Zxing application. The two bar charts have a very similar shape confirming the strong relationship between these two metrics. In particular, during the history, when lines of code increase or decrease, third-party invocations behave the same way.



**Figure 5.14.** The distribution of third-party calls of the Zxing application. The shape of this histogram is almost identical to the one depicted in Figure 5.13, that depicts the distribution of LOC of the same application.

This phenomenon is common to most of the applications we analyzed with *SAMOA*. Method invocations are one of the principal factors characterizing the size of apps (in terms of LOC). However, it remains to be investigated how this behavior influences the structure of the code of mobile applications and what is the real impact of method invocations on apps development.

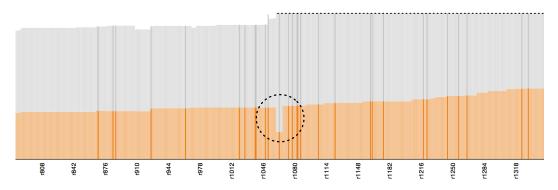
## 5.2.6 Out-of-Sync Manifest File

Scope: Distribution of the values of LOC.

**Description**: The Manifest file tells Android where to find the core elements of the application. Sometimes developers modify classes (*e.g.*, rename refactoring) without reflecting those changes in the manifest. The result is a temporary shrink of coreLOC due to missing or incorrect pointers listed on the manifest file. After some time, developers realize that the manifest is out-of-sync with the app and they bring it up to date. When the manifest is in synch, the coreLOC value returns to be correct.

**Symptoms**: The bar chart graph of the history presents a decreasing jump in the core LOC followed — some revisions later — by an increasing jump.

**Scenario**: Figure 5.15 shows part of the history of the Zxing app. At revision 1058 coreLOC drops from 674 to 356 while the value of total LOC is unchanged (*i.e.*, 1.8k). Some revisions later (*i.e.*, r1065) the core grows again to 700 LOC.



**Figure 5.15.** Out-of-sync manifest file of the Zxing application. The manifest became out of sync at revision 1058 when coreLOC drops. Seven revisions later, the manifest is updated and the value of coreLOC returns to be correct.

When we observed this behavior we hypothesized that it could be a linkage error in the Manifest file. In Android, developers must take care of manually updating the XML file that tells the system where to find the core classes (*i.e.*, with fully qualified names). In our experiences with the Android programming, forgetting to update the manifest was one of the major causes of bugs. In this case, the log message of revision 1058 says: *"Add history feature; group some functionality into subpackages*". We manually inspected the Manifest file of Zxing at different revisions discovering that, when the author reordered some functionalities into subpackages, he forgot to update the references in the manifest. A further evidence for this mistake is the commit comment of revision 1065: *"Unbroke the app after the big subpackage reshuffle of '09: Updated manifest entries [...]"*. It remains to be investigated how recurrent is this slip and what is its impact on Android development.

## 5.2.7 Snap Decrease of Core Elements

**Scope**: Distribution of the number of core elements.

**Description**: The history of the number of core elements presents snapshots in which the number of core elements decreases significantly.

**Symptoms**: The line chart shows consecutive data points (*i.e.*, number of core elements of a particular snapshot of an app) in which one point has a value that is more than two units smaller than the previous one.

**Scenario**: Figure 5.16 shows the history of the number of core elements of the Csipsimple application. In this app core elements decreases from 35 to 22 between revisions 304 and 305. From this point the number of core elements gradually increases to reach the value 35 at revision 1162. Suddenly, at revision 1163 there is a new snap decrease of core elements that lowers to 28. One should notice that in the remaining part of the history the number of core elements continues to decrease.

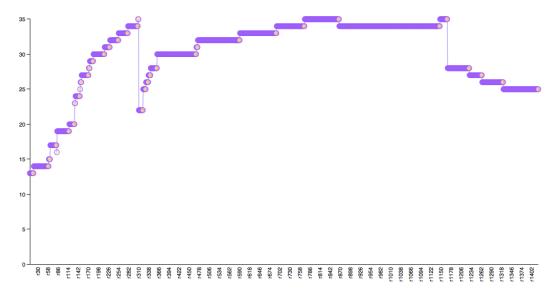


Figure 5.16. The distribution of the number of core elements of the Csipsimple application.

Since core elements are single tasks that the user can do in the application, their deletion is symptom of a reduction of functionalities. We suppose that this event is caused by refactoring activities. The log message of the first occurrence of this event says: *"Refactoring of accounts wizards Improve audio layer for 1.6 and lower (speaker issue)"*. With the support of additional sources of information (*e.g.*, source code and SVN log) we discovered that revision 305 affected more than 30 Java classes but there were no deletions. Investigating the Android Manifest we discovered where the deletions happened.

```
<activity android:name=".wizards.impl.Ekiga" </activity>
<activity android:name=".wizards.impl.Eutelia" </activity>
<activity android:name=".wizards.impl.Callcentric" </activity>
<activity android:name=".wizards.impl.Sip2Sip" </activity>
<activity android:name=".wizards.impl.Ippi" </activity>
<activity android:name=".wizards.impl.Pbxes" </activity>
<activity android:name=".wizards.impl.Pbxes" </activity>
<activity android:name=".wizards.impl.Pbxes" </activity>
<activity android:name=".wizards.impl.Pbxes" </activity>
<activity android:name=".wizards.impl.PlanetPhone" </activity>
<activity android:name=".wizards.impl.Sipgate" </activity>
<activity android:name=".wizards.impl.Phonzo" </activity>
<activity android:name=".wizards.impl.Pennytel" </activity>
</activity>
```

- ->

The listing above shows an excerpt of the manifest file at revision 305. Classes are not physically deleted but they were commented out from the XML file. The author also left a comment in the manifest saying that it *"should be refactored to get only one entry point"*. Our hypothesis is that, at the previous revision, the app was buggy and developers added this workaround preventing Android to reach the buggy classes. At revision 317 they solved the issue, *i.e.*, both the message and the commented entries in the manifest disappear.

Until revision 1162 the number of core element grows, then there is another snap decrease. This time the situation is different. In the commit comment the authors mention that they *"start UI refactoring [...] and drop the support for Android 1.5 to simplify the code.* In this revision the classes are physically deleted to perform perfective maintenance and drop the support with the legacy OS.

Our analysis helped us understanding the reasons behind these two snap decreases of core elements. It remains to be investigated whether the reduction of the number of core elements is always related to either perfective or corrective maintenance activities.

### 5.2.8 Snap Increase of Core Elements

**Scope**: Distribution of the number of core elements.

**Description**: The history of the number of core elements presents snapshots in which the number of core elements increases significantly.

**Symptoms**: The line chart shows consecutive data points (*i.e.*, number of core elements of a particular snapshot of an app) in which one point has a value that is more than two units bigger than the previous one.

**Scenario**: Figure 5.17 shows the history of the number of core elements of the Android VNC Viewer application. In this app core elements raise from 2 to 5 between revisions 16 and 17.

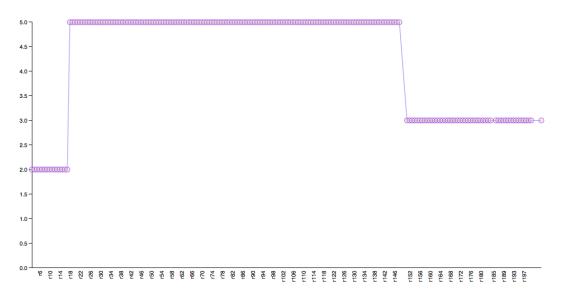


Figure 5.17. The distribution of the number of core elements of the VNC Viewer application.

Our intuition is that this phenomenon is due to to perfective maintenance activity, in which developers add a set of functionalities to their app. We investigated this application using other sources of data (*e.g.*, svn log) that confirmed our assumption. In fact, the log message of revision 17 says: *"Activities for listing, editing, viewing and deleting"*. Four new Activities were added while one was removed. It remains to be investigated if perfective maintenance is the only cause for a snap increase of core elements.

### 5.2.9 Stepwise Growth in History

**Scope**: Distribution of the values of LOC (and external calls).

**Description**: The distribution of LOC (or calls) presents a behavior like a stepwise function. **Symptoms**: The bar chart graph of the history presents remarkable increasing, or decreasing, variations in size.

**Scenario**: Figure 5.18 depicts the history of the Apps Organizer app. This system presents both increasing and decreasing jumps. Between revisions 33 and 34 the total LOC of the app jumps from 1.4k to more than 8k. One must note that — however — coreLOC does not increase much.

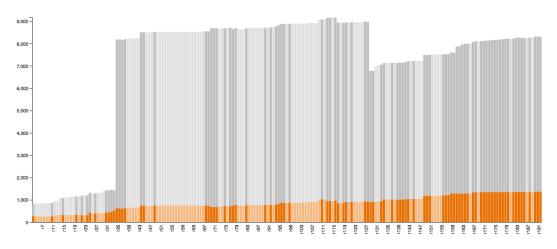


Figure 5.18. Jumps in the LOC distribution of Apps Organizer application.

We investigated on this issue using complementary pieces of data (*e.g.*, SVN logs). As we already mentioned, apps rely on external libraries. The surprising fact is that, sometimes, developers include the entire source code of some third-party APIs, as in the Apps Organizer example. At revision 34 the authors decided to increment the performance of their app by adding the Trove library<sup>2</sup>, that provides high speed collections for Java.

Jumps that significantly shrink the size of the app are also worth investigation. In this example, the decreasing jump between revision 127 and 128 is still related to the Trove library. In particular, the authors performed a refactoring and removed all the library source files that were not useful. It should be investigated if there are other common causes behind this jumps in the history.

<sup>&</sup>lt;sup>2</sup>See http://trove.starlight-systems.com/

### 5.2.10 Stepwise Increase of Core Elements

Scope: Distribution of the number of core elements.

**Description**: Over the history the number of core elements increases like a stepwise function. **Symptoms**: The line chart showing the distribution of the number of core elements presents intervals in which the number of core elements is constant and points in which it increases, often by a single unit.

**Scenario**: Figure 5.19 shows the history of the number of core elements of the Zxing app. This app presents 9 periods in which the number of core elements remains unchanged. Between these intervals the number of core elements raises (and in one case decreases).

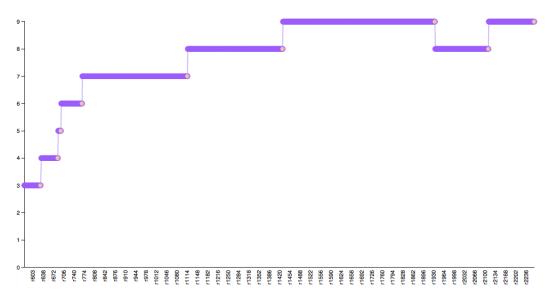


Figure 5.19. The distribution of the number of core elements of the Zxing application.

We observed this blueprint in a number of applications we analyzed. The factors to consider are (1) the number of intervals in which core elements are constant, (2) the duration of those periods, and (3) the events happening between the intervals (*i.e.*, addition or deletion of core elements. It remains to be investigated the impact of this evolutionary pattern, according to the factors listed before, on the development of mobile applications.

## 5.3 Summing Up

In this chapter we presented some characteristics of mobile applications that can be inferred using the views that we proposed. We divided the catalogue in peculiarities of single snapshots and in features that can be deduced by analyzing evolutionary pieces of data (*i.e.*, history). Our snapshot-based visualization helped us understanding both the importance of third-party method invocations and the relationships between core elements and non-core classes. In particular we discovered the following:

- Some apps are implemented entirely within core classes, *i.e.*, classes that inherits from Android base classes.
- Other apps present low core ratio, *i.e.*, the majority of classes are developed without extending base classes from the Android SDK.
- In some apps the core is composed of one (or a few) big classes (i.e., God classes).
- Some apps present a strict dominance of external method invocations, confirming our hypothesis that apps rely on third-party libraries.
- In some apps, instead, the majority of calls implement internal behaviors.

Visualizing the entire history of each app enables an higher level analysis. This unveils additional peculiarities of the development of mobile applications. In particular, we observed the following:

- Sometimes developers did not use versioning systems at the beginning of the development of apps. In this way the initial evolution of the software is lost.
- The evolution of the source code of apps often has idle periods. In these periods developers update the documentation or implement versions of the app for a different platform (*i.e.*, iOS).
- Sometimes the lines of code of apps vary substantially between consecutive revisions. The cause is often the addition to the project of the source code of third-party libraries.
- During refactoring developers often factor out some functionalities from core classes. This results in a significant decrease of coreLOC while the value of overall LOC remains constant (*i.e.*, core drop).
- Android developers have to manually maintain the manifest in sync with their source code. However, a common mistake is to not reflect source code changes (*e.g.*, rename refactoring) in the manifest file.
- Most of the times the distributions of the values of LOC and external method invocations present a similar shape. This represent an high correlation between these two software metrics and means that the size of apps grows together with the addition of external method calls.
- Some apps present a regular increase of the number of core elements, symptom of a constant addition of functionalities during the evolution.
- Other apps, instead, presents unanticipated additions (or deletions) of core elements. We discovered that, often, the reasons behind these events are maintenance activities.

# **Chapter 6**

# Conclusions

In this chapter we take a step back and summarize our work. Section 6.1 summarizes this thesis and retraces the contributions of our work. In Section 6.3 we propose some extensions to our research. Finally, Section 6.4 concludes the thesis.

## 6.1 Summary

Classical software systems were analyzed from various perspectives. For example, researchers proposed different approaches for software maintenance, software evolution, and program comprehension of traditional software systems [Leh96, LPR<sup>+</sup>97, LPR98, GJKT97, GT00, Tur02].

The introduction of the iPhone in 2007 started the *"Era of Mobile Apps"*. Apps are, for all purposes, software systems developed to run on specific mobile platforms. Unlike traditional software systems there is little material in the literature about mobile applications development. In this thesis we argued that apps differ from the software systems we are used to deal with.

To support our thesis, we have taken the first steps in this largely unexplored environment. After some preliminary analysis and feasibility studies we built *SAMOA*, a software analytics tool, customized for apps. Our tool supports the analysis of mobile applications through interactive visualizations at different granularities. Views are intuitive and useful means to assist analysts during their inspection tasks. We argue that apps development will remain a hot topic in the next years. To this aim, we made our work available to others in the form of an interactive web application in which users can benefit from our catalogue of visualizations.

We used the Overview Pyramid, proposed by Lanza & Marinescu [LM06], to initially characterize apps. With this approach we observed the first differences with traditional software systems, such as the the almost complete lack of inheritance, one of the leading object-oriented paradigms used in standard software systems. By studying the distribution of a chosen set of software metrics, we discovered an high correlation between the number of external method invocations and the McCabe Cyclomatic complexity number. This, together with the high correlation of lines of code and number of external calls, led us to claim that the size of an app, and its complexity, grow in correspondence with the addition of external method invocations. Under the hood, *SAMOA* employs a template, based on both software metrics and apps-specific concepts, to describe apps. We used different artifacts to extract software facts from apps. The most relevant are the AST and the MSE file, a generic format used to import models in Moose. In the last part of the thesis, we collected our findings in a catalogue of characteristics of apps.

Apps are, in all respects, software systems. This work is among the first attempts to study apps from a software engineering viewpoint. The study has extended our knowledge about apps under different perspectives. We discovered structural properties of apps and common habits among developers of apps. In particular, apps are significantly smaller than traditional software systems and strongly rely on third-party libraries. An interesting finding was to learn that developers used to include the source code of third-party libraries within the source code of their apps. Another insight regards the size of apps: sometimes a single class (*i.e.*, Activity) is enough to build up an entire application. Our evolutionary visualizations unveiled additional practices happening during the development of apps. Among them, we discovered that most of the times developers did not use versioning system from the beginning of the development and that often a single repository contained more than one apps or versions of the same app for different platforms. Moreover, we noticed that the most common error among apps' developers is forgetting to maintain up to date the synchronization between the manifest and the source code.

We just scratched the surface of the analysis of apps but our findings are promising and *SAMOA* provides great opportunities for subsequent work in this novel research field.

## 6.2 Limitations

The current approach has a number of limitations:

- **Single versioning system**. For this prototype we only considered applications using SVN to version their source code. This limitation cuts down our dataset because there are apps stored in F-Droid that use different versioning systems. Due to time constraints, we only built an SVN crawler but clients for different versioning systems can be developed.
- No real database. Our web-app retrieves JSON models using asynchronous JavaScript (*i.e.*, AJAX) requests since the files are stored in the server where the web-app is running. The JSON format was chosen in preparation for the usage of a schema-less database, such as CouchDB<sup>1</sup>. We did not spend efforts on this feature but in a future release this would be a priority. This prototype loads JSON using multiple AJAX requests. MapReduce queries with distributed views would surely decrease page load time and help solving the next issue.
- No permanent cache (only session-cache). This issue is related to the absence of a database. This prototype caches the data for the entire analysis session but when the user closes the browser the data is lost. We tried to implement permanent caching mechanisms but storing at the client the data we need was unfeasible due to current browsers' limitations (*i.e.*, max local storage is about 3MB against the 40MB that our tool needs).

62

<sup>&</sup>lt;sup>1</sup>See http://couchdb.apache.org/

• **Off-line back-end**. This prototype does not allow users to add their own repositories from the web-interface. The back-end is a Java program that produces the JSON files needed for the visualizations but is not implemented as a web-service. A future work is to wrap the back-end in a web-service and lets the user feed the repository URLs of the apps that he wants to analyze.

## 6.3 Future Work

This section proposes some extensions to our work. In this section we describe future research directions in this area as well as possible enhancements for our tool.

### 6.3.1 Support for Additional Platforms

Mobile applications are developed in a variety of languages. Each platform uses its own language: Android uses Java, Windows Phone 7 uses C#, and apps for iOS devices are written in Ojective-C. In our tool we only implemented support for Android applications (*i.e.*, Java). To investigate on apps of different sellers one have to implement the back-end of *SAMOA* for different programming languages. As described in Section 4.1, our back-end extracts metrics and measures from two artifacts, the Abstract Syntax Tree and the MSE file.

The requirements for this extensions are (1) the extraction & parsing of the AST, (2) the generation of the MSE file (the parsing can be done with our parser), and (3) the formulation of the *core of the app*.

For most of the languages listed above there are libraries to extract and traverse ASTs. The non-trivial part is the creation of the MSE files. MSE model exporters have been written for Java, C++, C#, and Smalltalk but are not available for other languages (*i.e.*, Objective-C). A developer could either create ad-hoc MSE exporters for these languages or devise an alternative strategy to extract the required metrics. Further investigation is needed to devise the core of apps developed for platforms different from Android.

### 6.3.2 Support for Additional Versioning Systems

In this prototype we focused on apps using SVN to handle code evolution. A possible extension consists in writing repository crawlers for versioning systems such as Git and SourceSafe. This would enlarge the mobile apps dataset providing more apps for the analysis.

### 6.3.3 Online Service

At this time, *SAMOA*'s back-end runs on a local machine and produces data (*i.e.*, JSON) that are manually uploaded on the web-server, where the web-app is running. For this reason, the user can only inspect apps already stored in our dataset.

A possible extension consists in wrapping the back-end into a web-service, and slightly modifying the user interface of the tool. In this way the newly created web-service allows users to specify the url of the repositories they want to analyze. *SAMOA* then asynchronously collects all the information required and notifies the user when his/her application is ready for analysis.

### 6.3.4 Better Data Storage Mechanism

Our tool stores JSON models in a remote folder where the web-application is running. To retrieve them uses AJAX requests. The JSON format was chosen in preparation for the usage of a schema-less database, such as CouchDB.

A possible extension is to replace those AJAX requests to the remote folder with queries to a database. This would both reduce page load time for apps with a long history and enhance our current cache mechanism.

#### 6.3.5 Enhancements to the Analysis

We only scratched the surface of apps analysis. Various approaches for software maintenance, software evolution and program comprehension were proposed for traditional software systems [Leh96, LPR<sup>+</sup>97, LPR98, GJKT97, GT00, Tur02]. It remains to be investigated, for example, if and how these techniques can be adapted to apps.

It could also be interesting to do a comparison between software facts extracted from apps and facts coming from traditional software systems. It is also possible to devise statistical thresholds to characterize apps, as Lanza & Marinescu did for traditional software systems [LM06].In addition, the analysis could be expanded in different directions. Considering bugrelated or contributors information are just two candidate factors for a more complete analysis.

Another possible extension is to combine source code data with rankings from Google Play. In this way we can estimate, if and to what extent, higher rankings eventually correspond to better code.

### 6.3.6 Enrichments of the Catalogue

Our catalogue of characteristics of apps collects some software facts discovered during our analysis. By expanding the analysis and considering additional factors (see Section 6.3.5), the catalogue can be enriched with apps peculiarities obtained from different perspectives.

For example, adding bug-related information to our analysis, could enrich the catalogue with unusual practices of apps developers while fixing bugs. Additionally, we could study contributors information and complement the catalog with interrelationships patterns among apps developers.

### 6.3.7 Validation with a Larger Dataset

In our study, we considered 20 applications. It remains to be investigated if using a larger dataset consolidates our findings. To this aim, a possible extension is to examine all apps in the F-droid repository.

# 6.4 Epilogue

In this thesis we have created a software analysis tool for mobile applications. Apps are a relatively new phenomenon, but the interest on the topic is rising. Recently, a paper aimed at understanding the reuse in the Android Market [RNAH12] has been accepted at the  $20^{th}$  IEEE International Conference on Program Comprehension<sup>2</sup>.

With *SAMOA* we scratched the surface of this new research field. We are just at the beginning. There are many steps to take in this novel environment, but our work is open to a wide number of extensions, as discussed in Section 6.3.

We believe that in the next years this area will become a hot research topic and will attract the interest of researchers.

<sup>&</sup>lt;sup>2</sup>See http://icpc12.sosy-lab.org/

# **Appendix A**

# **SAMOA: Implementation Details**

In this appendix we describe some implementation details. Section A.1 details the procedure used to collect and elaborate data while Section A.2 explains how the tool retrieves and visualize them.

## A.1 Data collection & export

The back-end of the tool prepares the data for the client. In this prototype, the data collection procedure takes place off-line (*i.e.*, on a local machine) and involves the following steps:

- 1. Read the list of repository URLs from a configuration file
- 2. For each URL:
  - (a) Checkout the repository at HEAD revision
  - (b) Abort if the repository structure has changed. Go to next URL (if any)
  - (c) Generate the Abstract Syntax Tree
  - (d) Extract NOP, NOC, NOM, and LOC
  - (e) Generate the MSE using VerveineJ
  - (f) Create an in-memory representation of the MSE
  - (g) Extract CYCLO, FANOUT, ANDC, AHH, INTC, and APIC
  - (h) Parse the Android Manifest (i.e., XML) using DOM parser
  - (i) Extract core elements and app version number
  - (j) Assigns values of LOC to core elements
  - (k) Generate a JSON file, including data coming from step 2d and 2g
  - (l) Update the repository to an earlier revision
  - (m) Go back to step 2b

In the rest of this section we describe the key steps in details.

#### Step 2b Abort if the repository structure has changed

The tool extracts source code information from two different sources, the AST and the MSE. We need to keep the two data origins in synch. For this reason we require that the source folder of the project is one of the immediate children of the repository URL that is fed to our SVN crawler. Suppose that the following address is provided to SAMOA: http://url.of.the. repository/svn/trunk/MyApplication/. Our tool expects to find the source folder within the directory that checks out, the "MyApplication" folder in this case. In this way we can ensure that our two extractors (*i.e.*, for AST and MSE) work on the same data, producing consistent results. In case the repository structure changes, the procedure stops and SAMOA would have available only data until the last snapshot checked out.

#### Steps 2c and 2d Generate and parse the Abstract Syntax Tree

The Eclipse JDT package provides facilities to create basic AST from source string. We used this technique to create the ASTs and we implemented a visitor to extract the software metrics we need. This approach works for high level metrics but unfortunately fails for more sophisticated measures, such as the distribution of method invocations. The reason for this is that bindings are not resolved if the AST is generated from a source string and the Eclipse environment is not set, as in our case. Binding information is obtained from the Eclipse Java model, but we do not have such model. To this aim, we used a parallel strategy to obtain other pieces of information.

#### Steps 2e, 2f and 2g Generate and scan the MSE file

MSE is a generic format used to describe any model and the best way to explain it is with an example<sup>1</sup>. The following snippet provides the representation of a small model using the MSE grammar.

```
( (FAMIX.Namespace (id: 1)
   (name 'aNamespace'))
 (FAMIX.Package (id: 201)
   (name 'aPackage'))
 (FAMIX.Package (id: 202)
   (name 'anotherPackage')
   (parentPackage (ref: 201)))
 (FAMIX.Class (id: 2)
   (name 'ClassA')
   (container (ref: 1))
   (parentPackage (ref: 201)))
 (FAMIX.Method
   (name 'methodA1')
   (signature 'methodA1()')
   (parentType (ref: 2))
   (LOC 2))
 (FAMIX.Attribute
   (name 'attributeA1')
   (parentType (ref: 2)))
 (FAMIX.Class (id: 3)
   (name 'ClassB')
```

 $<sup>^1\</sup>mathrm{This}$  example is taken from http://www.themoosebook.org/book/externals/import-export/mse

```
(container (ref: 1))
 (parentPackage (ref: 202)))
(FAMIX.Inheritance
 (subclass (ref: 3))
 (superclass (ref: 2)))
)
```

The file defines 8 entities: 1 Namespace, 2 Packages, 2 Classes, 1 Method, 1 Attribute and 1 Inheritance relation. Each entity has a unique identifier (*e.g.*, (id: 1)) and some properties. These properties can be either primitive, like (name 'aPackage'), or reference properties that point to other entities. As an example, (subclass (ref: 3)) denotes that the subclass of the inheritance relation is the class named 'ClassB'. Figure A.1 depicts the object graph of the model described by the MSE presented before.

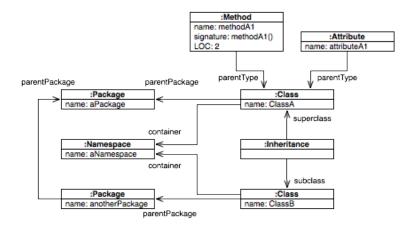


Figure A.1. The object graph of the simple MSE listed above.

To generate those MSE files there are dedicated exporters built for different languages. The two tools we tried were: *inFamix* and *VerveineJ*. The former is "a free C/C++ and Java parser and FAMIX model exporter, based on inFusion<sup>2</sup> technology". Unfortunately, after trying this tool, we discovered that it can not model library entities and defaults usages of such entities as *unknown*. Our aim was to model those entities so we had to try another exporter. VerveineJ, instead, can deal with library entities and allows the user to specify some useful command line options. In addition to some JVM parameters to increase the memory of the virtual machine, it allows to enumerate a class-path explicitly. We used this option to our advantage and we appended the Android SDK to the class-path. This allowed us to obtain MSEs with all the details we need for our analysis.

It happened that VerveineJ failed to produce an MSE. We did not investigate much on that and we just skipped the apps for which the exporter was not able to produce the model. This limited the dataset but since we further restricted the domain of the investigation, as explained in Section 2.4.2, this was not a significant issue.

<sup>&</sup>lt;sup>2</sup>See http://www.intooitus.com/products/infusion

We built a parser that uses a scanner to identify the entities in the MSE files (*i.e.*, tokenizes the MSE). This approach allows us to parse each MSE once and construct an in-memory representation of each file. The remaining work consists in extracting, from the representation we created, the software metrics needed (*e.g.*, CYCLO, FANOUT, ANDC, AHH, INTC, and APIC). **Steps 2h, 2i and 2j** *Processing the Android Manifest* 

In step 2d *SAMOA* extracted software metrics for all classes in the project. Steps 2h and 2i consist in scanning the Android manifest and extracting the useful pieces of information. Core elements are among this information. The following snippet shows part of an Android manifest.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
package="ch.inf.android.app"
android:versionCode="1"
android:versionName="1.0">
<application android>
   <activity
      android:name=".TheActivity"
      android:label="@string/title_info">
      <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
   </activity>
   <activity
      android:name=".AnotherActivity"
      android:theme="@android:style/Theme.Dialog"
      android:label="@string/title_details" />
   <activity
      android:name=".YetAnotherActivity"
      android:launchMode="singleTop"
      android:label="@string/title_configure"
      android:windowSoftInputMode="stateHidden">
   </activity>
</application>
```

The relevant parts of the Manifest are (1) definitions of core elements, with their properties, (2) the package declaration, and (3) the android:versionName. This example, shows 3 core elements (*i.e.*, 3 activities) named TheActivity, AnotherActivity and YetAnotherActivity. In this case, TheActivity is the main activity as defined in its intent-filter (*i.e.*, android.intent.action.MAIN). The package declaration is used to determine the fully qualified name of the classes appearing in the manifest. The attribute called 'package' identifies its name (*i.e.*, "ch.inf.android.app"). The last important factor that our parser collects is the version of the app, identified by the tag 'android:versionName'.

Android Manifests declare classes in different ways. For example, it happens that the package declaration is absent or that some definitions refer to classes declared outside the project. There is no connection with names appearing in the manifest and the physical Java files that contain the classes. For this reason, we devised different mechanisms to reconstruct the fully qualified name of the classes (*i.e.*, using the package declaration, whenever possible) and pair this definitions with their values of the LOC metric obtained in step 2d.

## A.2 Data retrieval & visualization

The outcome of the back-end is a set of JSON files representing multiple snapshots of a series of applications. This files are the input for our visualizations.

The front-end leverages a range of technologies, including HTML/CSS, PHP and JavaScript. Our web-application is composed by a single HTML page that is updated, using JavaScript, to reflect users' needs. The main functionalities are detailed in the following list.

### 1. List the available apps in our dataset

When the user visits the page for the first time *SAMOA* executes some PHP scripts to obtain the list of available apps in the ecosystem. For this prototype, we did not use a database even though it is one of features listed in our future work. JSON files are stored in a remote folder where our web-application is running. PHP scripts traverse the content of this folder and presents to the user the list of available applications.

### 2. For each app, list the available revisions (i.e., snapshots)

For each application we stored multiple snapshots. We developed some PHP scripts that run while the page loads for the first time and collect, for each application, all the available revisions. We cached this data in a JavaScript object (*i.e.*, a map) that associates, to each app, the list of available snapshots so that PHP runs only once.

### 3. Retrieve a list of JSON files to visualize

Visualizations require JSON files. Each file represents a snapshot of an app. As explained in Section 4.2.1, even single-app visualization use historical data to determine their proportions. To this aim, to visualize an application, *SAMOA* needs its complete history. For this reason, when a particular app is requested, the tool triggers a series of AJAX requests and gathers the complete history of the app.

### 4. Cache the data for the current session

Unfortunately, the procedure described at point 3 requires some time, depending on how much data the tool requires. For this reason, we implemented a cache mechanism that stores the data locally for the analysis session. In this way if during the analysis the user requires twice the same data, the second time, data is loaded without delay.

#### 5. Visualize the data

Our views are built using  $d3.js^3$  "a JavaScript library for manipulating documents based on data". We implemented different scripts that receive as input one or more JSON files and dynamically construct the views. Once visualizations are presented to the user he/she can interact with them, as described in Section 4.4.

<sup>&</sup>lt;sup>3</sup>See http://d3js.org/

# A.3 Summing Up

In this appendix we detailed how the back-end produces the data and how the client-side retrieves them to produce the visualizations. In this prototype, data are produced on a local machine. The procedure starts when *SAMOA*'s back-end checks out the SVN repositories of apps and, using two different approaches (*i.e.*, AST and MSE), extracts the set of software metrics it requires. The outcome of this step is a set of JSON files (*i.e.*, one for each snapshot) that are loaded to our web-service. From this moment on, our web-application can fetch data to produce visualizations. The client-side leverages a range of technologies (*e.g.*, HTML/CSS, PHP, and JavaScript) to obtain the data, dynamically modify its web-page page and compose visualizations. In this chapter, we also discussed the problems we encountered and the strategies we adopted to solve them.

# **Appendix B**

# **Additional Details on the Dataset**

During this thesis we named various mobile applications. In this chapter we report (1) the links to their websites and (2) the locations of their repositories. Notice that we present pieces of information about the 20 apps that compose our final dataset *i.e.*, see Section 2.4.2.

Name	Website address
Alogcat	http://code.google.com/p/alogcat/
Andless	<pre>http://code.google.com/p/andless/</pre>
Android VNC Viewer	<pre>http://code.google.com/p/android-vnc-viewer/</pre>
Anstop	<pre>http://code.google.com/p/anstop/</pre>
App Soundmanager	<pre>http://code.google.com/p/app-soundmanager/</pre>
Appsorganizer	<pre>http://code.google.com/p/appsorganizer/</pre>
Csipsimple	<pre>http://code.google.com/p/csipsimple/</pre>
Diskusage	http://code.google.com/p/diskusage/
Mythdroid	http://code.google.com/p/mythdroid/
Mythmote	http://code.google.com/p/mythmote/
Open GPSTracker	<pre>http://code.google.com/p/open-gpstracker/</pre>
Opensudoku	<pre>http://code.google.com/p/opensudoku-android/</pre>
Replicaisland	<pre>http://code.google.com/p/replicaisland/</pre>
Ringdroid	http://code.google.com/p/ringdroid/
Search Light	<pre>http://code.google.com/p/search-light/</pre>
Share My Position	<pre>http://code.google.com/p/sharemyposition</pre>
Sipdroid	http://code.google.com/p/sipdroid/
Solitaire for Android	<pre>http://code.google.com/p/solitaire-for-android/</pre>
Zirco Browser	http://code.google.com/p/zirco-browser/
Zxing (Barcode scanner)	http://code.google.com/p/zxing/

Table B.1. Links to the websites of apps.

iewer ager	http://alogcat.googlecode.com/svn/trunk/
iewer ager	
iewer ager	http://andless.googlecode.com/svn/trunk/
ager	http://android-vnc-viewer.googlecode.com/svn/trunk/eclipse_projects/androidVNC/
ager	http://anstop.googlecode.com/svn/trunk/
	http://app-soundmanager.googlecode.com/svn/trunk/
	http://appsorganizer.googlecode.com/svn/trunk/Apps0rganizer/
	http://csipsimple.googlecode.com/svn/trunk/CSipSimple/
	http://diskusage.googlecode.com/svn/trunk/
Mythdroid http://	http://mythdroid.googlecode.com/svn/trunk/
	http://mythmote.googlecode.com/svn/trunk/
Open GPSTracker http://	http://open-gpstracker.googlecode.com/svn/trunk/OpenGPSTracker/trunk/application/
	http://opensudoku-android.googlecode.com/svn/trunk/OpenSudoku/
Replicaisland http://	http://replicaisland.googlecode.com/svn/trunk/
Ringdroid http://	http://ringdroid.googlecode.com/svn/trunk/
	http://search-light.googlecode.com/svn/trunk/
Share My Position http://	http://sharemyposition.googlecode.com/svn/trunk/ShareMyPosition-android/
Sipdroid http://	http://sipdroid.googlecode.com/svn/trunk/
Solitaire for Android http://	http://solitaire-for-android.googlecode.com/svn/trunk/
Zirco Browser http://	http://zirco-browser.googlecode.com/svn/trunk/
Zxing (Barcode scanner) http://	http://zxing.googlecode.com/svn/trunk/android/

Table B.2. Repository URLs of our dataset.

# **Bibliography**

- [BAB<sup>+</sup>93] David M. Butler, J. C. Almond, R. D. Bergeron, K. W. Brodlie, and A. B. Haber. Visualization reference models. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 337–342, San Jose, CA, October 1993. IEEE Computer Society Press.
- [BAHS97] Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.
- [CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [Dav95] Alan Mark Davis. 201 Principles of Software Development. McGraw-Hill, 1995.
- [DBL10] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In Ji Wang, W. K. Chan, and Fei-Ching Kuo, editors, Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010, pages 23–31. IEEE Computer Society, 2010.
- [DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. A pattern language for reverse engineering. In *Proceedings of EuroPLoP '2000*, pages 189–208. UVK GmbH, 2000.
- [GJKT97] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In Proceedings International Conference on Software Maintenance (ICSM'97), pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press.
- [GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *IWPSE*, pages 113–122. IEEE Computer Society, 2005.
- [GT00] Michael Godfrey and Qiang Tu. Evolution in open source software: A case study. In Proceedings International Conference on Software Maintenance (ICSM 2000), pages 131–142, Los Alamitos CA, 2000. IEEE Computer Society Press.
- [HB08] Reid Holmes and Andrew Begel. Deep intellisense: a tool for rehydrating evaporated information. In Proceedings of the 2008 international working conference on Mining software repositories, MSR '08, pages 23–26, New York, NY, USA, 2008. ACM.

[HHM04]	Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. Msr 2004: International workshop on mining software repositories. In <i>ICSE</i> , pages 770–771, 2004.
[HJZ12]	Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: MSR for app stores. <i>International Collegiate Programming Contest</i> , 2012.
[IIM10]	Rashedul Islam, Rofiqul Islam, and Tohidul A. Mazumder. Mobile Application and Its Global Impact. <i>International Journal of Engineering &amp; Technology</i> , 10, December 2010.
[Lan01]	Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In <i>Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)</i> , pages 37–42, 2001.
[Lan03a]	Michele Lanza. Codecrawler — A lightweight software visualization tool. In <i>Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)</i> , pages 51–52. IEEE CS Press, 2003.
[Lan03b]	Michele Lanza. Object-Oriented Reverse Engineering — Coarse-grained, Fine- grained, and Evolutionary Software Visualization. PhD thesis, University of Bern, May 2003.
[Leh85]	M. M. Lehman. <i>Program Evolution: Processes of Software Change</i> , chapter 12, pages 247–274. Academic Press, London, UK, 1985.
[Leh96]	Manny Lehman. Laws of software evolution revisited. In European Workshop on Software Process Technology, pages 108–124, Berlin, 1996. Springer.
[LK94]	Mark Lorenz and Jeff Kidd. <i>Object-oriented software metrics - a practical guide</i> . Prentice Hall, 1994.
[LLGR10]	Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The Small Project Observatory: Visualizing software ecosystems. <i>Science of Computer Programming</i> , 75(4):264–275, April 2010.
[LM06]	Michele Lanza and Radu Marinescu. <i>Object-Oriented Metrics in Practice</i> . Springer-Verlag, 2006.
[LPR <sup>+</sup> 97]	Manny Lehman, Dewayne Perry, Juan Ramil, Wladyslaw Turski, and Paul Wernick. Metrics and laws of software evolution–the nineties view. In <i>Proceedings IEEE In-</i> <i>ternational Software Metrics Symposium (METRICS'97)</i> , pages 20–32, Los Alamitos CA, 1997. IEEE Computer Society Press.
[LPR98]	M. M. Lehman, Dewayne E. Perry, and Juan F. Ramil. Implications of evolution metrics on software maintenance. In <i>ICSM</i> , page 208, 1998.
[Lun09]	Mircea Lungu. Reverse engineering software ecosystems. PhD thesis, University of Lugano, Switzerland, October 2009.
[mar10]	marketsandmarkets.com. Global mobile application market (2010-2015). Markets and Markets, August 2010.

- [McC76] T. J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
- [Neu11] Sylvie Neu. Telling evolutionary stories with complicity. Master's thesis, Faculty of Informatics of the University of Lugano, 2011.
- [NLHD11] Sylvie Neu, Michele Lanza, Lile Hattori, and Marco D'Ambros. Telling stories about GNOME with complicity. In *VISSOFT*, pages 1–8. IEEE, 2011.
- [PBS93] Blaine A. Price, Ronald Baecker, and Ian S. Small. A principled taxonomy of software visualization. J. Vis. Lang. Comput, 4(3):211–266, 1993.
- [Rie96] Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.
- [RNAH12] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Understanding reuse in the android market. *International Collegiate Programming Contest*, 2012.
- [SMWH09] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 23–33, Washington, DC, USA, 2009. IEEE Computer Society.
- [Som89] Ian Sommerville. Software Engineering. Addison-Wesley, third edition, 1989.
- [SW89] J. A. Simpson and Weiner. *The Oxford English Dictionary*. Oxford: Oxford University Press, 2nd edition, 1989.
- [TA08] Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. *Comput. Graph. Forum*, 27(3):831–838, 2008.
- [Tur02] Wladyslaw M. Turski. The reference model for smooth growth of software systems revisited. *IEEE Trans. Software Eng*, 28(8):814–815, 2002.
- [WL08] Richard Wettel and Michele Lanza. Codecity: 3D visualization of large-scale software. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume, pages 921–922. ACM, 2008.