Software Analytics for Mobile Applications – Insights & Lessons Learned

Roberto Minelli and Michele Lanza
REVEAL @ Faculty of Informatics — University of Lugano, Switzerland

***Abstract*—Mobile applications, known as *apps*, are software systems running on handheld devices, such as smartphones and tablet PCs. The market of apps has rapidly expanded in the past few years into a multi-billion dollar business. Being a new phenomenon, it is unclear whether approaches to maintain and comprehend traditional software systems can be ported to the context of apps.**

We present a novel approach to comprehend apps from a structural and historical perspective, leveraging three factors for the analysis: source code, usage of third-party APIs, and historical data. We implemented our approach in a web-based software analytics platform named SAMOA.

We detail our approach and the supporting tool, and present a number of findings obtained while investigating a corpus of mobile applications. Our findings reveal that apps differ significantly from traditional software systems in a number of ways, which calls for the development of novel approaches to maintain and comprehend them.

***Keywords*-mobile applications; software evolution, maintenance, analytics; mining software repositories;**

I. Introduction

Mobile applications, also known as *apps*, are software systems aimed at smartphones, tablet PCs, and other handheld devices. Apps are implemented in programming languages usually dictated by the platform: Java for Android, Objective-C for iOS, C# for Windows phone, etc. Each vendor provides its own distribution channel (*e.g.,* Google Play for Android apps, App Store for iOS apps). The apps marketplace is vast: The Apple and Android stores, for example, offer around one million apps for download. Islam *et al.* [1] affirm that the development of apps is having significant impact both from an economical and from a social perspective. They reported that the apps business generated a revenue of ca. \$4.5 billion USD in 2009. Markets & Markets predict that the global apps business will be worth \$25 billion USD in 2015 [2]. As the popularity of apps increases, maintaining them will become critical.

Like traditional software systems, apps evolve over time and require maintenance activities. Classical approaches to software maintenance and program comprehension [3], [4], [5], [6], [7], [8] were developed when apps did not exist, and it is unclear if those approaches can be ported to apps. Apps are distributed through app stores that do not provide source code, as mentioned by Harman *et al.* [9]. To overcome this problem, we settled on a public catalogue of FOSS (free and open source) apps for the Android platform, named F-Droid[1].

We present an in-depth investigation of a large corpus of apps from a structural and historical perspective. Our analysis focuses on three factors: (1) source code, (2) usage of third-party Application Programming Interfaces (APIs), and (3) historical data. We want to answer questions such as: How does an app differ from a traditional system in terms of size and complexity? Do apps make intensive use of third-party APIs? Does the source code of apps contain the usual code smells [10] or are there smells specific to apps? To support our analysis, we developed a software analytics platform for apps: Samoa [11]. Samoa mines software repositories of apps and uses visualizations to present the data. We present a number of findings obtained while investigating the F-Droid corpus. For example, we noticed that the use of inheritance is essentially absent in apps, that apps heavily rely on external APIs, and that most apps are short-lived single developer projects. In this article[2] we make the following contributions:

- An in-depth analysis of apps from a structural and historical perspective.
- A presentation of Samoa, a web-based software analytics tool for apps.
- A collection of insights pertaining to the maintenance and comprehension of apps.

Structure of the Paper: In Section II we review related work. In Section III we detail our approach and Samoa, our supporting tool. In Section IV we present our findings. In Section V we summarize our work and outline future work.

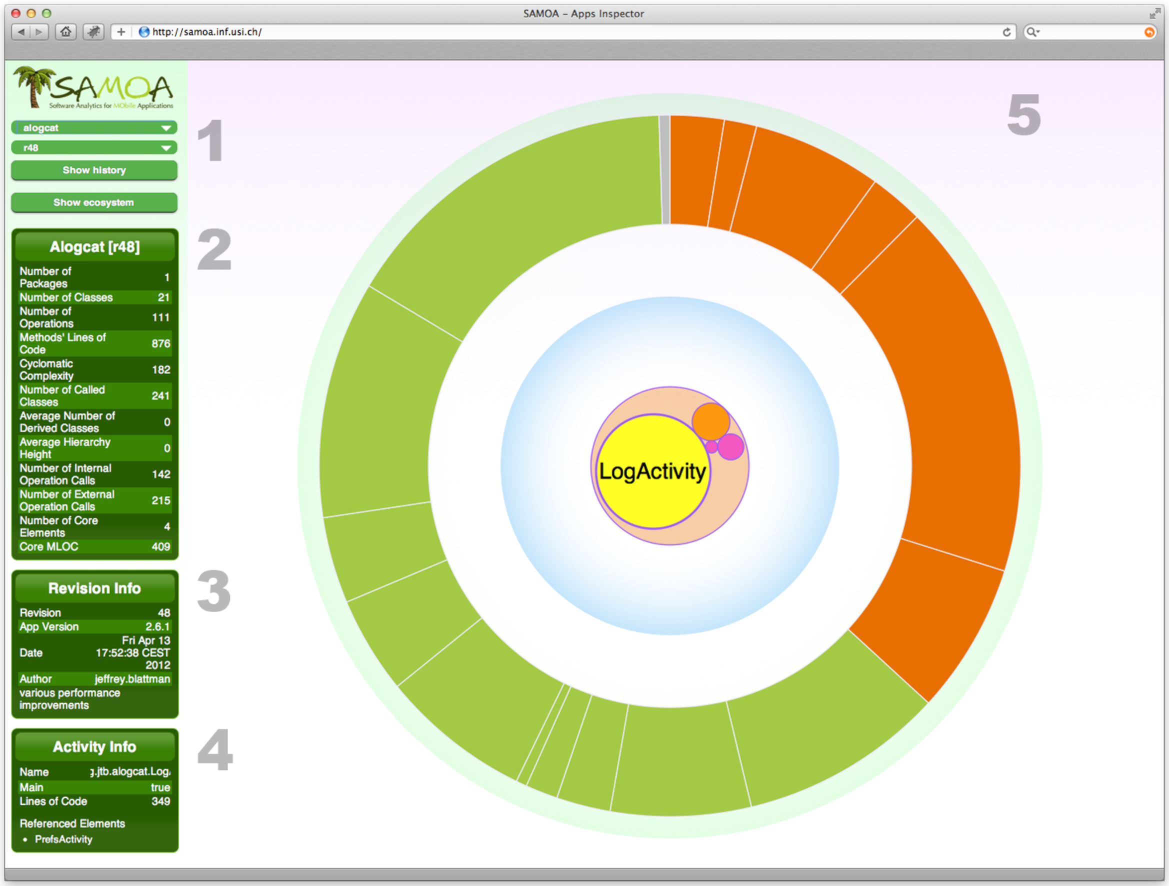
II. Related Work

Due to the recency of apps, there is little directly related work, and its nature is quite heterogeneous.

Ruiz *et al.* explored software design aspects of apps, focusing on reuse by inheritance and class reuse [12]. They divided apps in categories (*i.e.,* Cards & Casino, Personalization, Photography, Social and Weather), and found that 61% of all classes in each category appear in two or more apps. Hundreds of apps were completely reused by another app in the same category. Harman *et al.* introduced App Store Mining [9], and focused on aspects pertaining to factors of success of apps with respect to their distribution channels. We want to study the source code of apps, rather than their channels of distributions, to understand if and how they differ from traditional software systems, and which are the possible implications for the maintenance of apps.

[1]See http://f-droid.org/

[2]For understandability we advise reading a color version of this paper.



Alogcat [r48]

Number of Packages	1
Number of Classes	21
Number of Operations	111
Methods' Lines of Code	876
Cyclomatic Complexity	182
Number of Called Classes	241
Average Number of Derived Classes	0
Average Hierarchy Height	0
Number of Internal Operation Calls	142
Number of External Operation Calls	215
Number of Core Elements	4
Core MLOC	409

Revision Info

Revision	48
App Version	2.6.1
Date	Fri Apr 13 17:52:38 CEST 2012
Author	jeffrey.blattman

various performance improvements

Activity Info

Name	g.jtb.alogcat.Log
Main	true
Lines of Code	349

Referenced Elements

- PrefsActivity

Figure 1: A first glance of SAMOA.

Indirectly related are studies of the Android operating system, which is in itself a traditional software system, *i.e.,* the following works do not focus on apps, but on the framework that allows apps to be built. Khomh *et al.* [13] conducted an empirical study to understand how Android adapts the Linux kernel, from which it is derived. Asaduzzaman *et al.* [14] mapped bug reports and changes to identify bug-introducing changes. Martie *et al.* [15] used statistical models to identify Android's most debated high-level issues. Reina and Robles [16] analyzed Android to understand who is involved in localization and translation activities. Guana *et al.* [17] used Android bug repository data to comprehend the architectural layers of Android. Hu *et al.* [18] compared Android's concrete and conceptual architecture. Sinha *et al.* [19] performed a quantitative investigation of Android's change history.

Apps have only recently started to be the focus of software engineering research, this explains the small amount of related work. At this time it is uncertain whether classical software maintenance and program comprehension approaches can be used for apps, which motivates the present work.

III. SAMOA

To support our analysis we developed SAMOA, an interactive web-based visual software analytics platform to analyze apps from a structural and historical perspective. SAMOA is available at http://samoa.inf.usi.ch. The User Interface (UI) of SAMOA is divided into 5 parts (see Figure 1):

1) **Selection panel.** It allows the user to pick the app to be analyzed, and to switch between the three different interactive visualizations offered by SAMOA.
2) **Metrics panel.** It displays a set of metrics for a specific revision of an app chosen through the selection panel. At ecosystem level, it displays global measurements.
3) **Revision info panel.** It displays information about a specific revision of an app (*i.e.,* snapshot).
4) **Entity panel.** It displays data about the entity in focus.
5) **Main view.** The main surface dedicated to the interactive visualizations: (1) a snapshot view to depict a specific revision of one app, (2) a history view to depict the evolution of one app, and (3) an ecosystem view to depict more apps at once.

A. Software Metrics for Apps

We gather, through a module of Samoa, a set of metrics for our app analysis. They are listed in Table I.

Metric	**Description**	**Scope**
NOP	*The* **Number of Packages** *of a project.*	A
NOC	*The* **Number of Classes** *defined by the user.*	**A**P
NOM	*The* **Number of Methods** *defined by the user.*	AP**C**
LOC	*The number of (non-empty)* **Lines of Code**.	**A**P**CM**
CYCLO	*McCabe's* **Cyclomatic Complexity** *[20].*	APC**M**
CALLS	*The number of (distinct)* **Method Calls**.	AP**CM**
FANOUT	*The* **Number of Called Classes** *[21].*	AP**CM**
ANDC	*The* **Average Number of Derived Classes** *[22]. This metric does not count interfaces.*	**A**P
AHH	*The* **Average Hierarchy Height** *of a system. A class is a root class if it is not an interface and not derived from user-defined classes.*	**A**P
INTC	*The* **Number of Internal Calls**, i.e., *invocations of methods that implement internal behavior.*	**A**PCM
EXTC	*The* **Number of External Calls**, i.e., *invocations that refer to third-party libraries.*	**A**PCM
NOCE	*The* **Number of Core Elements**, i.e., *classes composing the core of the app.*	**A**P
CoreLOC	*The sum of* **LOC of core elements**.	**A**P
Commits	*The* **Number of Commits** *of an app.*	**A**PC
CALLR	*The* **ratio between INTC and EXTC**.	**A**PCM
CORER	*The* **ratio between CoreLOC and LOC**.	**A**PCM

Table I: Software metrics for apps.

The scope of a metric denotes to which type of entity (*i.e.,* App, Package, Class, or Method) it pertains. Some finer-grained metrics, *e.g.,* LOC (Lines of Code), can be used at any granularity level through aggregation, but in practice we use them for specific scopes, indicated in bold.

B. Visualizing Apps

Samoa provides three visualizations to explore apps: a *snapshot view* to depict a specific revision of one app, a *history view* to depict the evolution of one app, and an *ecosystem view* to depict several apps at once.

Snapshot View

Figure 2 depicts our snapshot view. It presents the essential structural properties of an app using a circular view depicting the core of an app (*i.e.,* the classes) and the external API calls it makes. Core elements are the entities specific to the development of apps (*i.e.,* inheriting from the mobile platform SDK's base classes). In Android apps, they are specified in the manifest (*i.e.,* An XML file that presents essential information about an app). The view depicts each core element as a circle, where its radius is proportional to the value of LOC, and the color indicates its type (*e.g.,* Activity, Service, Main Activity). Figure 2 depicts the Alogcat[3] app.

[3]See http://code.google.com/p/alogcat/

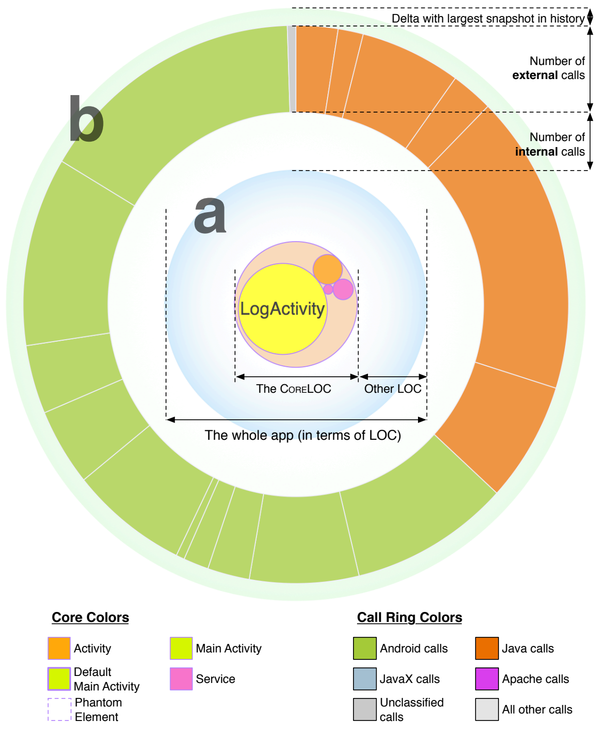


Figure 2: A snapshot view of Alogcat.

Alogcat is composed of 21 classes. The central part of the visualization (Figure 2.a) presents information about the size of the app (in terms of LOC). The core of the app accounts for nearly half the size of the app (*i.e.,* 409 CoreLOC out of 876 LOC) and is composed of four elements: two services, a main activity, and a default main activity. The remaining 17 classes are not depicted, but their LOC value is represented in Figure 2 as the difference between the radiuses of the app (*i.e.,* blue shaded circle) and the core circle (*i.e.,* red). The call ring (Figure 2.b) depicts third-party method invocations. Its thickness is proportional to the number of external method calls. Each portion of the ring represents calls to a distinct third-party library. Colors distinguish calls to different libraries. The angle spanned by an arc is proportional to the number of API calls. Alogcat mostly calls two libraries: Android SDK (*i.e.,* green) and Java (*i.e.,* orange). The outer radius of the call ring indicates the size of an app, considering both LOC and method calls. The green shaded circle represents the maximum size of an app over its history. Figure 2 shows a gap between the green circle and the outer radius of the call ring, meaning that the current revision is not the largest in the app's history. The snapshot view provides also the means to assess the ratio between internal and external method calls. The thickness of the white ring between the app and the call ring portrays the number of internal calls. In Alogcat the call ring is wider than the white ring, meaning there are more calls to third-party libraries than calls to methods of user-defined classes.

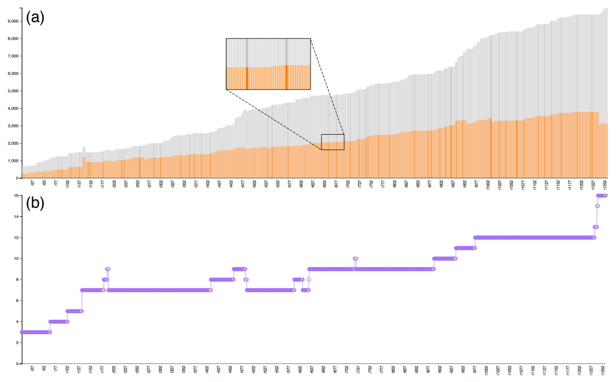


Figure 3: Evolution views of Open-GPSTracker in terms of (a) LOC and (b) number of core elements.

Evolution View

The *evolution view* (see Figure 3) depicts, using stacked bar charts and line charts, evolutionary information (*e.g.,* LOC, external calls, or core elements) about an app, depicting all the snapshots available to Samoa. Each bar represents a snapshot of an app, divided into layers, according to the type of data presented. The height of each bar represents the value of a specific software metric. The view uses opacity to denote an app's release versions: Darker bars are snapshots whose release number changed. Figure 3 shows the evolution of LOC of the Open-GPSTracker[4] app. The layers are CoreLOC (*i.e.,* red) and non-CoreLOC (*i.e.,* grey), and the height of each bar portrays the value of LOC. We also use line charts to represent data without a logical layer subdivision, such as the number of core elements presented in Figure 3.b.

Ecosystem View

The *ecosystem view* depicts several apps at once, using stacked bar charts or a grid view (see Figure 4). In the grid view each element is a simplified snapshot view: The radius of the core (*i.e.,* yellow) corresponds to the number of CoreLOC, the total radius is proportional to the total number of LOC, the span of the call ring shows the proportions of external calls, but omits the number of internal calls.

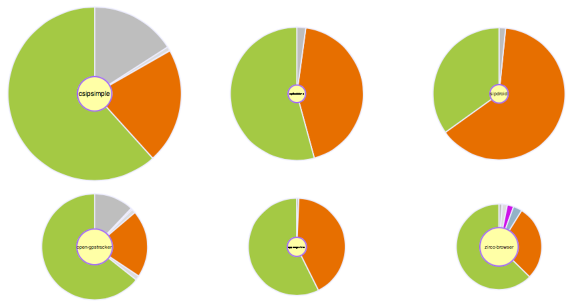


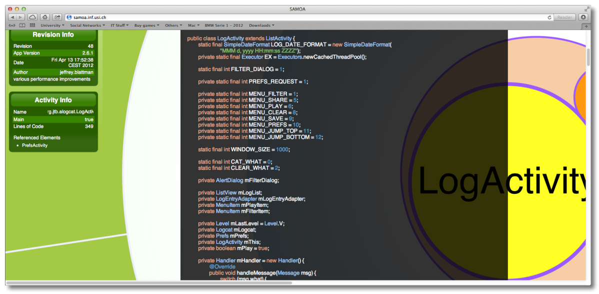
Figure 4: An ecosystem view of 6 apps.

[4]See http://code.google.com/p/open-gpstracker/

Figure 4 shows 6 apps, sorted according to their LOC size. This view is useful to get a "big picture" of several apps and then drill down using the snapshot and the evolution view.

User Interaction

In Samoa all visualizations are interactive. For example, by hovering on a shape, the entity is highlighted and the user is provided with additional information about that entity. The user can use the mouse to pan the snapshot view. Scrolling (*i.e.,* mouse wheel) on the metrics table (Figure 1.2) allows to change the zoom level. On clicking on a core element, Samoa shows its source code, as depicted in Figure 5.



Revision Info

Revision	48
App Version	2.6.1
Date	Fri Apr 13 17:52:38 CEST 2012
Author	jeffrey.blattman

various performance improvements

Activity Info

Name	rg.jtb.alogcat.LogActi
Main	true
Lines of Code	349

Referenced Elements

- PrefsActivity

```
public class LogActivity extends ListActivity {
    static final SimpleDateFormat LOG_DATE_FORMAT = new SimpleDateFormat(
            "MMM d, yyyy HH:mm:ss ZZZZ");
    private static final Executor EX = Executors.newCachedThreadPool();

    static final int FILTER_DIALOG = 1;

    private static final int PREFS_REQUEST = 1;

    private static final int MENU_FILTER = 1;
    private static final int MENU_SHARE = 5;
    private static final int MENU_PLAY = 6;
    private static final int MENU_CLEAR = 8;
    private static final int MENU_SAVE = 9;
    private static final int MENU_PREFS = 10;
    private static final int MENU_JUMP_TOP = 11;
    private static final int MENU_JUMP_BOTTOM = 12;

    static final int WINDOW_SIZE = 1000;

    static final int CAT_WHAT = 0;
    static final int CLEAR_WHAT = 2;

    private AlertDialog mFilterDialog;

    private ListView mLogList;
    private LogEntryAdapter mLogEntryAdapter;
    private MenuItem mPlayItem;
    private MenuItem mFilterItem;

    private Level mLastLevel = Level.V;
    private Logcat mLogcat;
    private Prefs mPrefs;
    private LogActivity mThis;
    private boolean mPlay = true;

    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
```

Figure 5: Samoa displaying the source code of Alogcat.

In the bar charts data can be re-ordered and layers can be grouped or stacked. In both the evolution and ecosystem view clicking on a shape leads to a snapshot view. To illustrate how user interaction is supported in Samoa we provide a screencast located at http://samoa.inf.usi.ch/samoa.mov.

C. Samoa *Behind the Curtains*

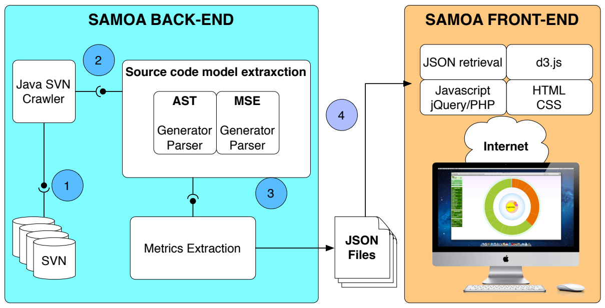


Figure 6: Architectural overview of Samoa.

1) Architectural Overview: Samoa has a back-end and a front-end, as Figure 6 depicts. The back-end is responsible for (1) mining apps-specific data (source code and history) data from the repository; (2) processing the data, by parsing the source code (analyzing both the Abstract Syntax Tree and the MSE[5]); (3) extracting a set of software metrics from the AST and the MSE file; and (4) generating JSON[6] files that are provided to the front-end.

[5]See http://www.moosetechnology.org/docs/mse
[6]See http://www.json.org

2) Modeling apps: To model apps and their history we extended classical software concepts (structure, source code, revisions, *etc.*) with concepts specific to mobile applications. In an app, we distinguish two sets of classes: *"core"* and *"non-core"*. The former are entities specific to the development of apps (*i.e.,* classes that inherit from the mobile platform SDK's base classes), and the latter are the remaining classes. In Android, core classes are Activities and Services, and are specified in the manifest file. Also, since apps extensively rely on third-party libraries, we explicitly model not only invocations of methods within the app, but also of methods located in external third-party APIs.

IV. Insights about Mobile Applications

We present insights that pertain to apps, in terms of maintenance and evolution-related issues and which have an impact on the applicability of existing analysis approaches. Our analysis is based on the F-Droid corpus[7], *i.e.,* a catalogue of FOSS apps for Android. To conduct an in-depth analysis we restricted the dataset to a final corpus composed of the 20 apps listed in Table II.

Name	Rate	Installs	Start rev.	End rev.	Size (LOC)
Alogcat	4.6	>100k	2	48	876
Andless	4.2	>100k	2	93	2'372
Android VNC	4.3	>1m	2	203	4'949
Anstop	N/A	N/A	2	61	1'142
AppSoundmanager	4.5	>50k	1	157	1'605
Appsorganizer	4.6	>1m	3	191	8'321
Csipsimple	4.4	>100k	2	1415	20'777
Diskusage	4.7	>50k	2	69	4'749
Mythdroid	N/A	N/A	76	640	6'114
Mythmote	4.6	>10k	2	281	1'593
Open GPSTracker	4.2	>100k	2	1255	9'754
Opensudoku	4.6	>1m	15	415	3'813
Replicaisland	4.2	>1m	2	7	14'192
Ringdroid	4.6	>10m	2	66	3'516
Search Light	4.7	>100k	2	4	272
Share My Position	4.6	>10k	2	76	468
Sipdroid	4.0	>500k	50	620	14'019
Solitaire	4.3	>10m	2	30	3'343
Zirco Browser	3.8	>10k	65	457	6'429
Zxing	4.3	>50m	569	2257	3'407

Table II: The final corpus.

Rating and installs information come from the Google play store[8] (where "N/A" means that the app is not listed in the store). The first and last snapshot available to Samoa, are reported in the columns "start" and "end revision".

[7]See http://f-droid.org/
[8]See https://play.google.com/store

During our analysis, we discovered peculiarities of Android apps. We devised an extensive catalogue [11] of use cases, symptoms, and possible scenarios, which we cannot present here due to space reasons. Instead, we present a subset of observations supported by a description, one or more examples, and a discussion of the implications.

Apps are smaller than traditional software systems.

Description: In terms of LOC, apps are small compared to traditional software systems. Often a few classes compose an entirely working app. In our dataset, the average size is 5.6 kLOC. The smallest app, Searchlight[9], has less than 300 LOC; the largest app, Csipsimple[10], has ca. 20 kLOC.

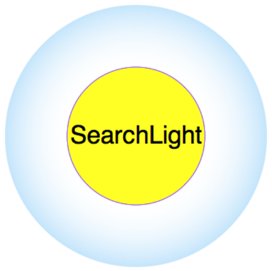


Figure 7: The SearchLight application.

Example: Figure 7 depicts the Searchlight application, which is composed of two classes: a main activity (*i.e.,* the yellow entity), and a non-core class (whose LOC value is proportional to the remaining part of the core container, the blue shaded circle). Together, they sum up to 272 LOC.

Implications: Many apps have a small set of functionalities, thus a few classes are enough to build them. Nevertheless it seems they are not trivial to maintain, as we see with the next observation.

Apps are inherently complex, mostly because they rely on third-party libraries.

Description: In the subset of apps of the F-Droid corpus we analyzed, external calls make up roughly 2/3 of all method invocations. In many apps external calls represent more than 75% of the total number of method invocations. Consequently, the number of calls implementing internal behavior is small. This is visible from our snapshot visualization: The thickness of the call ring represents the number of third-party calls. The number of internal calls is associated with the thickness of the white ring between the shaded blue circle and the call ring. Comparing these two measures gives an indication about the ratio of external and internal method calls.

[9]See https://code.google.com/p/search-light/
[10]See http://code.google.com/p/csipsimple/

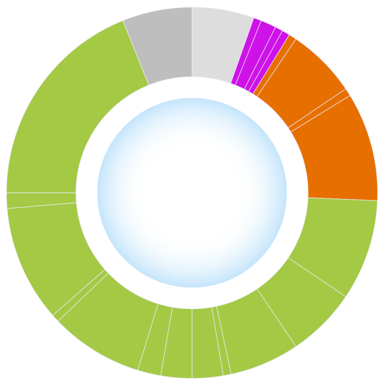


Figure 8: The call ring of Share My Position application.

Example: Figure 8 shows the Share My Position[11] application, which has 157 external invocations and 48 calls implementing internal behavior.

Implications: Since apps heavily rely on external libraries, in addition to look at their source code to comprehend them, one must also understand the behavior of the employed third-party libraries, complicating program comprehension and maintenance [23], [24], [25], [26].

The size and complexity of apps grow in correlation with the addition of third-party method invocations.

Description: In our analysis, we studied a set of software metrics of apps and their relationships. We observed that Pearson's linear correlation coefficient between number of external calls and McCabe's cyclomatic complexity number is high. The correlation between number of LOC and external calls is also strong. On the entire dataset, the average values for these correlations are respectively 0.82 and 0.84.

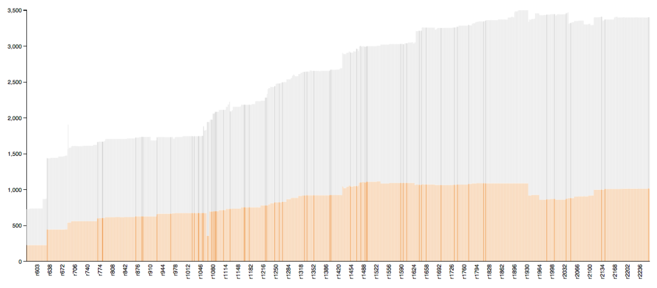


Figure 9: The evolution of LOC of the Zxing application.

Example: Figure 9 and 10 respectively show the evolution of number of LOC and external calls of the Zxing[12] app. The two bar charts have analogous shapes, supporting our observation: During the history, when LOC increase or decrease, third-party invocations behave the same way.

[11]See http://code.google.com/p/sharemyposition/
[12]See http://code.google.com/p/zxing/

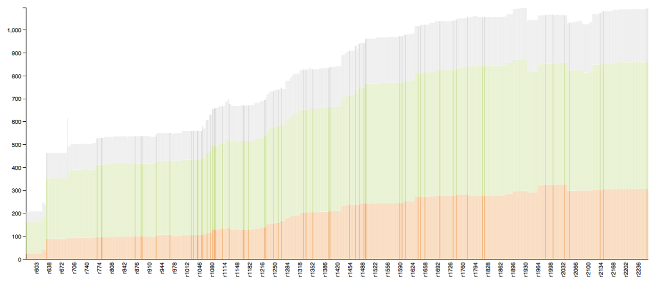


Figure 10: The evolution of third-party calls of Zxing.

Implications: We observed high values for the correlations of EXTC vs. CYCLO, and EXTC vs. LOC. Since the behavior of apps relies to a large extent on external libraries, the growth of an app is causally connected with the usage of the external libraries.

The use of inheritance is essentially absent in apps.

Description: Since Android apps are object-oriented systems, we also studied the use of inheritance employing two software metrics: Average Hierarchy Height (AHH) and Average Number of Derived Classes (ANDC) [22].

Example: The apps in our corpus have very small average values for both these metrics (ANDC = 0.19, AHH = 0.09) compared to traditional Java systems [22].

Implications: On the one hand the near-absence of inheritance could be justified by the fact that apps tend to be small, and thus there is little potential for inheritance. However, apps are real world systems, and we suspect that if they evolve over a long time, they will grow in size, according the Lehman's software evolution laws [27].

On the other hand, it could also point to the fact that many apps are not developed in a systematic way, and when programming is performed in a sluggish way, inheritance is a likely victim. This might be a problem, since inheritance helps to better structure the code and enables code reuse. Daly *et al.* analyzed traditional software systems and showed that on average, maintaining a flat system requires 20% more effort than an analogous system using inheritance [28].

Some apps contain the entire source code of third-party libraries.

Description: Android apps are Java systems, where usually third-party source code is reused by referencing JAR (Java ARchive) files containing the byte code of the external library. Developers of apps have a tendency of directly importing the entire source code of third-party libraries instead of adding the needed JAR files to their projects.

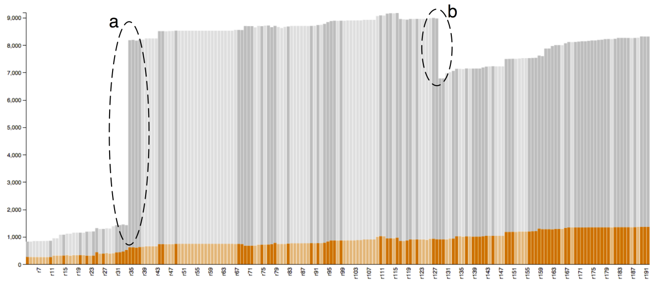


Figure 11: The evolution of LOC of Apps Organizer.

Example – Evolution view: Figure 11 depicts the evolution of the number of LOC of the Apps Organizer[13] application. The view presents remarkable increases and decreases in terms of size. In Figure 11.a the authors added the source code of the Trove library[14] that provides high speed collections for Java. Later on, as depicted in Figure 11.b, they removed some unused classes of the same library.

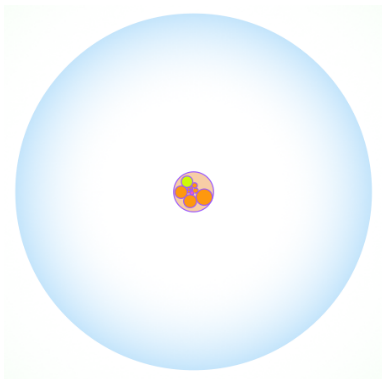


Figure 12: The Sipdroid application.

Example – Snapshot view: Figure 12 depicts the Sipdroid[15] application. The app has ca. 14k LOC, but only 1.5k are CoreLOC. The app is a Voice Over IP client and uses JSTUN[16] *i.e.,* a Java-based library for Simple Traversal of User Datagram Protocol through Network Address Translation. Developers included all the source code of that library.

Implications: One implication is that mobile devices still possess only limited computing resources, and code bloat in such a context is not desirable [29], [30]. We believe this to be a minor problem, since mobile devices are getting more and more powerful. The major implication is that copying external code into a system can have a series of far from obvious legal consequences [31], [32], which are probably underestimated by many apps developers.

[13]See http://code.google.com/p/appsorganizer/
[14]See http://trove.starlight-systems.com/
[15]See http://code.google.com/p/sipdroid/
[16]See http://jstun.javawi.de/

Some developers use versioning systems only at later stages of the development.

Description: Developers of apps seem to ignore an established common-sense practice: To put software projects under revision control early on. On average, the LOC of the first revision represents ca. one third of the LOC at the end of the evolution, which points to the fact that the first versions of the system were never under revision control.

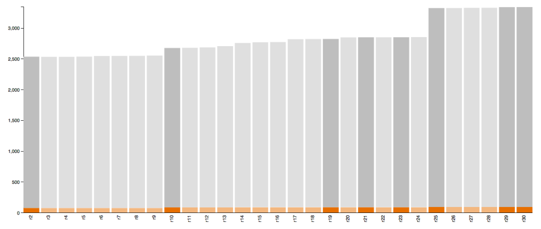


Figure 13: The evolution of LOC of Solitaire for Android.

Example: Figure 13 shows the evolution of LOC of Solitaire for Android[17]. At the beginning of the history the app has already 2.5 kLOC. We assume the app had a previous evolution, not versioned, or versioned elsewhere. The SVN log confirms our conjecture: *"Initial add, corresponds to market version 1.8"*.

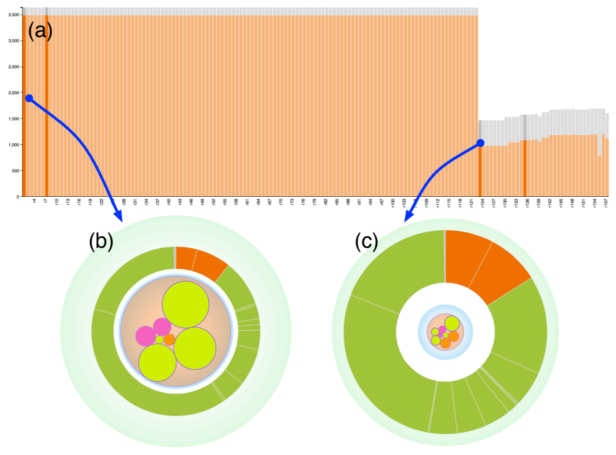


Figure 14: a) Evolution view of App-SoundManager (LOC); b) Snapshot view of rev. 1; c) Snapshot view of rev. 123.

Example: Figure 14.a depicts the evolution of LOC of App-SoundManager[18]. In the first 122 revisions (Figure 14.b), apparently, the app is left unchanged. A deeper analysis revealed that developers worked on one of the *"branches"* of the repository, *i.e.,* schedules. At revision 123 (Figure 14.c) they *"merge from schedules branch"*. However, we do not have any means to reconstruct what happened.

[17]See http://code.google.com/p/solitaire-for-android/
[18]See http://code.google.com/p/app-soundmanager/

Implications: The quality of any software analysis is connected to the quality of the available data, as otherwise wrong conclusions can be drawn [33], [34]. Incomplete histories of apps make retrospective software evolution analysis difficult.

Developers often break the connection between Android manifest and source code.

Description: The manifest file tells Android where to find core elements within an app. Developers must manually maintain this file in sync with the source code. Sometimes developers modify classes (*e.g.,* rename refactoring) without reflecting the changes in the manifest.

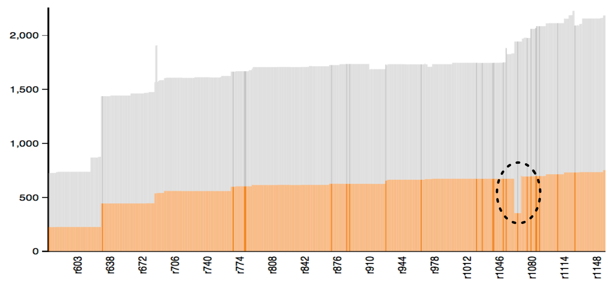


Figure 15: The evolution of LOC of the Zxing app.

Example: Figure 15 shows part of the evolution of the LOC of Zxing. The figure highlights what we call "*core drop*": The number of CoreLOC decreases, while the overall number of LOC remains constant. In Zxing the authors reordered some functionalities into sub-packages, but they forgot to update the references in the manifest. Then, at later revisions they *"unbroke the app after the big subpackage reshuffle of '09: Updated manifest entries [. . .]"*.

Implications: Like any software system, apps are also made up of pieces that are not source code [35], but necessary for their functioning. We believe that as apps will get more complex the maintenance and understanding of such pieces will become a concern.

Development guidelines are often ignored.

Description: Software systems should conform to a set of sound guiding principles. For example, the Android documentation states that *"an app consists of multiple activities loosely bound to each other. Typically,* **one activity** *is specified as the "main" activity, which is presented to the user when launching the application for the first time"*. Activities are depicted as core elements in our snapshot view. We observed many apps have more than one main activity. Android lets one specify also a "default main activity": The real activity to invoke when the app is started. In some apps there are also multiple occurrences of such special activities.

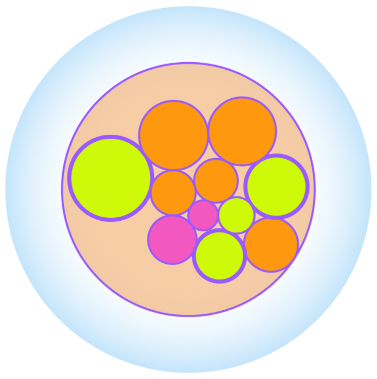


Figure 16: The App-SoundManager application. Main activities are yellow, *"default"* main activities have a thicker stroke.

Example: Figure 16 depicts a snapshot of App-SoundManager. The core has eight Activities (*i.e.,* orange and yellow) and two Services (*i.e.,* purple). Four of the activities are labeled as "main" and three of them are "default" main activities (*i.e.,* thicker stroke in our snapshot view).

Implications: Multiple main activities represent diverse entry points for apps, which complicates their comprehension.

Some apps are only composed of the core.

Description: On average, CoreLOC represent roughly half of the size of an entire app. The snapshot visualization unveils that some apps have almost only CoreLOC. In our corpus 25% of the apps have more than 70% of CoreLOC.

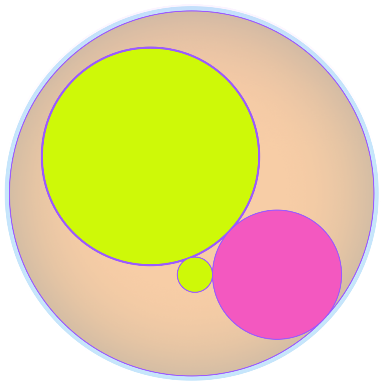


Figure 17: The AndLess application.

Example: Figure 17 shows the AndLess[19] application, which has about 2.3k LOC, and only 60 of those are non-CoreLOC. The snapshot view shows a small gap between the core circle (*i.e.,* light red circle) and the entire app (*i.e.,* shaded blue circle).

[19]See http://code.google.com/p/andless/

Implications: Apps entirely implemented within core element violate basic design guidelines regarding separation of concerns and encapsulation. For example, AndLess is a music player composed only of four classes. Its main activity (*i.e.,* the big yellow circle in Figure 17) counts 1.7k LOC and it is responsible for (1) drawing the UI, (2) starting & stopping the music, (3) recursively traversing the file system to find the music, (4) parsing playlists, (5) handling CUE files, *etc.* The main activity is de facto a "god activity", and like god classes represents a maintenance problem [36].

V. Conclusions

The first apps were born when Apple introduced the iPhone, back in 2007. Apps are software systems, and like traditional software systems, they evolve and require maintenance activities. Since traditional approaches to program comprehension and maintenance [3], [4], [5], [6], [7], [8] were created before apps appeared, it is unclear whether they can be used or adapted to the context of apps.

We performed an in-depth investigation of a corpus of Android apps, supported by our freely available software analytics platform named Samoa. Some of our findings are unique to Android apps because they involve app-specific concepts (*e.g.,* Android manifest, core elements). Other insights could possibly be applied to traditional systems as well, but further investigation is required. We discovered that apps present substantial differences to classical software systems. In the first place, apps are significantly smaller: The average value of LOC for the apps in our corpus is 5.6k. Nevertheless it seems they are not trivial to comprehend and maintain, since the behavior of apps relies to a large extent on external libraries (*i.e.,* on average, external calls account for about 2/3 of all method invocations). Their comprehension involves the understanding of the employed third-party libraries, in addition to the source code of apps, complicating maintenance activities [23], [24], [25], [26].

Nowadays, apps are smaller, simpler, and have less functionality than traditional software systems, but we believe this difference is destined to disappear. On the one side this implies that in the future apps will be more powerful and will have a richer set of features. In the long term, on the other hand, since the distinction between apps and traditional systems will become subtle, apps will face the same software maintenance and program comprehension issues faced with classical systems. Our investigation points out that while many existing approaches should be portable to the context of mobile applications they may need to be tailored to the specific context of apps.

Future Work. We analyzed a set of open-source Android apps with fairly short histories. As part of our future work we want to investigate long lived apps. We also plan to investigate a larger dataset to confirm the insights obtained so far and to collect new ones. Moreover, we did not consider apps for platforms other than Android. Apps for iOS, for example, could present different peculiarities or structural properties. As part as our future work, we want to add support for additional platforms (*e.g.,* iOS, Windows Phone, Nokia) and therefore also extend our supporting tool, Samoa. It remains to be seen how we can leverage the differences between apps and traditional software systems to derive novel approaches to maintain and comprehend apps.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "GSync" (SNF Project No. 129496). We thank Alberto Bacchelli for his feedback on this work.

References

[1] R. Islam, R. Islam, and T. Mazumder, "Mobile application and its global impact," *International Journal of Engineering & Technology (IJEST)*, 2010.

[2] Markets and Markets, "Global mobile application market (2010–2015)," 2010.

[3] M. Lehman, "Laws of software evolution revisited," in *Proceedings of EWSPT 1996 (5th European Workshop on Software Process Technology)*. Springer, 1996, pp. 108–124.

[4] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution - the nineties view," in *Proceedings of METRICS 1997 (4th International Symposium on Software Metrics)*. IEEE Computer Society Press, 1997, pp. 20–32.

[5] M. Lehman, D. Perry, and J. Ramil, "Implications of evolution metrics on software maintenance," in *Proceedings of ICSM 1998 (14th International Conference on Software Maintenance)*. IEEE Computer Society Press, 1998, p. 208.

[6] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, "Software evolution observations based on product release history," in *Proceedings of ICSM 1997 (13th International Conference on Software Maintenance)*. IEEE Computer Society Press, 1997, pp. 160–166.

[7] M. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proceedings of ICSM 2000 (16th International Conference on Software Maintenance)*. IEEE Computer Society Press, 2000, pp. 131–142.

[8] W. Turski, "The reference model for smooth growth of software systems revisited," *Transactions on Software Engineering (TSE)*, pp. 814–815, 2002.

[9] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: MSR for app stores," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE Computer Society Press, 2012.

[10] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] R. Minelli, "Software analytics for mobile applications," Master's thesis, University of Lugano, 2012.

[12] I. Ruiz, M. Nagappan, B. Adams, and A. Hassan, "Understanding reuse in the android market," in *Proceedings of ICPC 2012 (20th IEEE International Conference on Program Comprehension)*. IEEE Computer Society Press, 2012, pp. 113–122.

[13] F. Khomh, H. Yuan, and Y. Zou, "Adapting linux for mobile platforms: An empirical study of android," in *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)*. IEEE CS Press, 2012, pp. 629–632.

[14] M. Asaduzzaman, M. Bullock, C. Roy, and K. Schneider, "Bug introducing changes: A case study with android," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE CS Press, 2012, pp. 116–119.

[15] L. Martie, V. Palepu, H. Sajnani, and C. Lopes, "Trendy bugs – topic trends in the android bug reports," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE CS Press, 2012, pp. 120–123.

[16] L. Reina and G. Robles, "Mining for localization in android," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE Computer Society Press, 2012, pp. 136–139.

[17] V. Guana, F. Rocha, A. Hindle, and E. Stroulia, "Do the stars align? multidimensional analysis of androidś layered architecture," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE Computer Society Press, 2012, pp. 124–127.

[18] W. Hu, D. Han, A. Hindle, and K. Wong, "The build dependency perspective of android's concrete architecture," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE Computer Society Press, 2012, pp. 128–131.

[19] V. Sinha, S. Mani, and M. Gupta, "Mince: Mining change history of android project," in *Proceedings of MSR 2012 (9th Working Conference on Mining Software Repositories)*. IEEE Computer Society Press, 2012, pp. 132–135.

[20] T. McCabe, "A measure of complexity," *Transactions on Software Engineering (TSE)*, pp. 308–320, 1976.

[21] M. Lorenz and J. Kidd, *Object-oriented Software Metrics*. Prentice Hall, 1994.

[22] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[23] S. Raemaekers, A. van Deursen, and J. Visser, "Exploring risks in the usage of third-party libraries," in *BENEVOL 2011 (10th BElgian-NEtherlands software eVOLution)*, 2011.

[24] ——, "An analysis of dependence on third-party libraries in open source and proprietary systems," in *Proceedings of CSMR 2012 (16th European Conference on Software Maintenance and Reengineering)*, 2012, pp. 64–67.

[25] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)*. IEEE Computer Society Press, 2012, pp. 483–492.

[26] V. Bauer and L. Heinemann, "Understanding api usage to support informed decision making in software maintenance," in *Proceedings of CSMR 2012 (16th European Conference on Software Maintenance and Reengineering)*, 2012, pp. 435–440.

[27] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, no. 9, pp. 1060–1076, 1980.

[28] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "The effect of inheritance on the maintainability of object-oriented software: an empirical study," in *Proceedings of ICSM 1995 (11st International Conference on Software Maintenance*. IEEE Computer Society Press, 1995, pp. 160–166.

[29] N. Mitchell and G. Sevitsky, "The causes of bloat, the limits of health," in *Proceedings of OOPSLA 2007 (22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2007, pp. 245–260.

[30] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/FoSER 2010 (18th International Symposium on Foundations of Software Engineering – Workshop on Future of Software Engineering Research*. ACM Press, 2010, pp. 421–426.

[31] M. Di Penta, D. Germán, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the evolution of software licensing," in *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, 2010, pp. 145–154.

[32] D. Germán, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proceedings of MSR 2009 (6th Working Conference on Mining Software Repositories)*, 2009, pp. 81–90.

[33] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of ESEC/SIGSOFT FSE 2009 (7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, 2009, pp. 121–130.

[34] D. Kawrykow and M. Robillard, "Non-essential changes in version histories," in *Proceedings of ICSE 2011 (29th International Conference on Software Engineering)*, 2011, pp. 351–360.

[35] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan, "An empirical study of build maintenance effort," in *Proceedings of ICSE 2009 (31st International Conference on Software Engineering)*, 2011, pp. 141–150.

[36] A. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.