Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Suelen Goularte Carvalho
Orientador: Prof. Dr. Marco Aurélio Gerosa

Instituto de Matemática e Estatísticas
Universidade de São Paulo
Programa de Ciência da Computação

12 de Dezembro de 2016

Sumário

Cheiros de código

- Cheiros de código são sintomas de design ruim e más práticas de programação [13, 18, 33, 74].
- São conhecidos também pelos termos:
  - más práticas (anti-patterns) [78],
  - anomalias [54] e
  - maus cheiros (bad smells) [33].
- Nesta pesquisa usamos o termo cheiro de código para nos referenciar a ambos os termos más práticas [78], anomalias [54] e maus cheiros de código [33].

Camada de apresentação Android

Camada de apresentação Android é o termo definido nesta pesquisa para delimitar o objeto de estudo e se refere aos seguintes elementos de um projeto Android:

- Activities e Fragments,
- Listeners,
- Recursos da Aplicação como:
  - XMLs de layout,
  - XMLs de estilos,
  - Arquivos gráficos (imagens e XMLs gráficos),
  - dentre outros.
- Adapters.

Contexto

- Sobre desenvolvimento de software:
  - Código fácil de manter e evoluir é importante [Gibbon, 1997].
  - Existem diversas heurísticas, práticas e padrões amplamente conhecidas para escrever código "limpo" [16, 18, 36, 50].
  - Existem diversas ferramentas que usam métricas de qualidade de código existentes para identificar códigos problemáticos [11, 12].
- Sobre o Android:
  - É a maior plataforma móvel da atualidade, com mais de 83% de fatia de mercado, e vem crescendo ano após ano [8, 14].
  - Projetos Android apresentam diferenças com relação a projetos tradicionais, como Java [39, 46, 60].
  - Diferenças principalmente com relação à GUI [17, 40, 71].

Motivação

- Acadêmica:
    - Aniche (2016) [18] em sua tese de doutorado abordou um tema similar porém relacionado ao framework Spring MVC.
    - Em 2016 a Google Play Store superou 200 milhões de aplicativos disponíveis [10].
    - Manutenibilidade de código Android é um tema ainda em aberto [39].
- Pessoal:
    - Possuo 6 anos de experiência com a plataforma Android.
    - Pela minha experiência, a pesquisa aborda problemas do dia-a-dia de desenvolvimento Android.

Problema

- Heurísticas e métricas de código existentes são pouco contextuais [18].
- A falta de heurísticas para detectar trechos de código problemáticos específicos sobre Android impossibilita:

  (i) a definição de métricas de qualidade de código,

  (ii) a implementação de ferramentas de detecção automática desses trechos problemáticos.

Objetivo

Identificar boas e más práticas no contexto da camada de apresentação de aplicativos Android nativos e, com base nessas práticas, definir e validar um catálogo de cheiros de código.

Questões de pesquisa

- Questão geral:

  Quais cheiros de código são específicos a camada de apresentação Android?

- Questões específicas:
  - **QP1:** O que desenvolvedores consideram boas e más práticas com relação a camada de apresentação em projetos Android?
  - **QP2:** Qual a relação entre os cheiros de código propostos e a tendência a mudanças e defeitos?
  - **QP3:** Desenvolvedores Android percebem os códigos afetados pelos cheiros de código propostos como problemáticos?

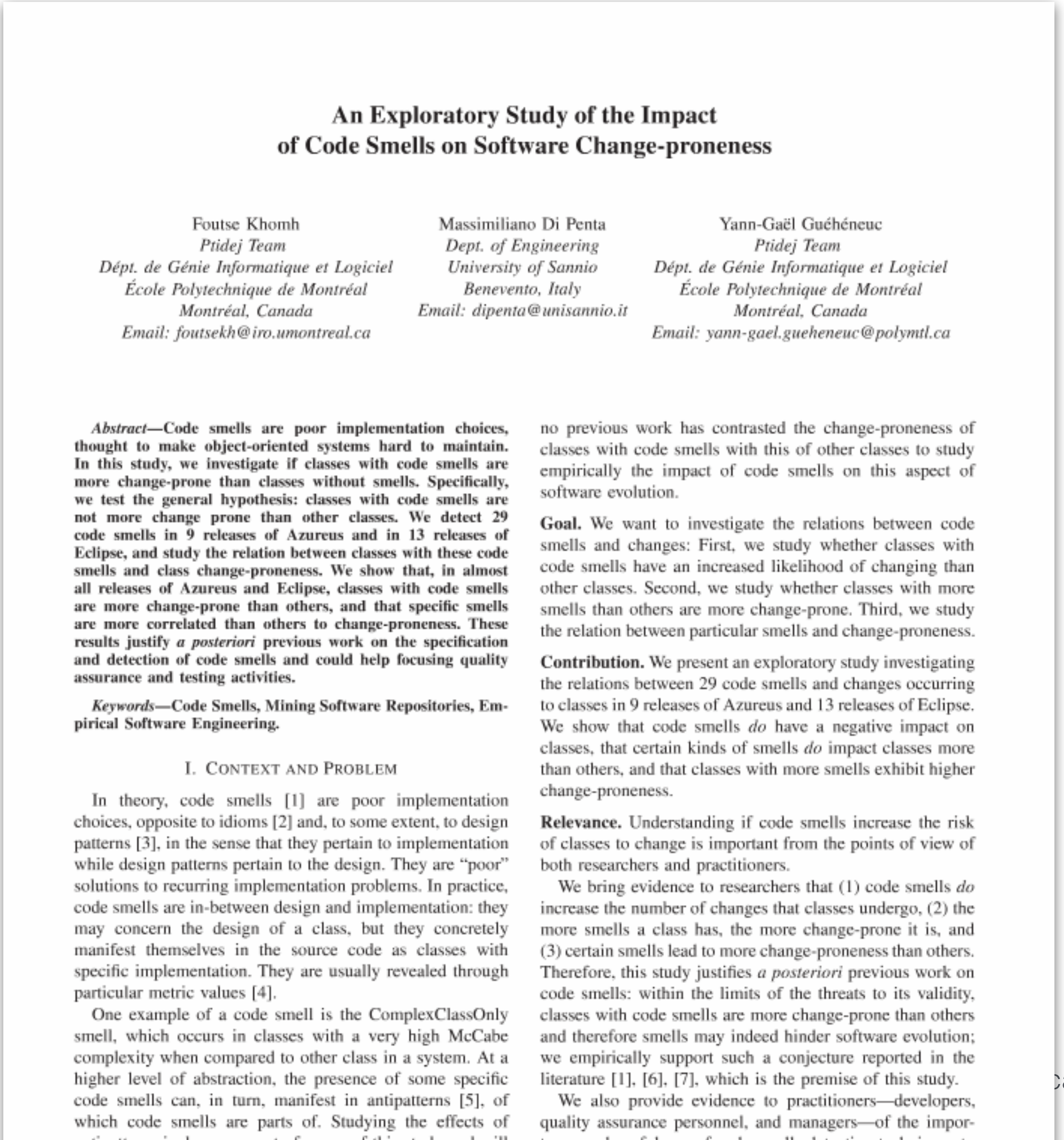
Contribuições esperadas

- Catálogo validado de cheiros de código na camada de apresentação Android.
- Estudo quantitativo sobre a tendência à mudanças e defeitos dos cheiros de código propostos.
- Estudo qualitativo sobre a percepção de desenvolvedores Android com relação aos cheiros de código propostos.

Fontes e estruturação

- Fontes: ACM, IEEE Xplore, Google Scholar, Google e outros.
- Termos (português e respectivos em inglês): anomalias, cheiros de código, Android, mau cheiro e móvel.
- Categorizamos os trabalhos relacionados em 4 grupos:
    - Pesquisas relacionadas a cheiros de código.
    - Pesquisas relacionadas à plataforma Android.
    - Cheiros de código específicos à tecnologia ou plataforma.
    - Cheiros de código relacionados à plataforma Android.

Pesquisas relacionadas a cheiros de código



An Exploratory Study of the Impact of Code Smells on Software Change-proneness

Foutse Khomh
Ptidej Team
Dépt. de Génie Informatique et Logiciel
École Polytechnique de Montréal
Montréal, Canada
Email: foutsekh@iro.umontreal.ca

Massimiliano Di Penta
Dept. of Engineering
University of Sannio
Benevento, Italy
Email: dipenta@unisannio.it

Yann-Gaël Guéhéneuc
Ptidej Team
Dépt. de Génie Informatique et Logiciel
École Polytechnique de Montréal
Montréal, Canada
Email: yann-gael.gueheneuc@polymtl.ca

***Abstract*—Code smells are poor implementation choices, thought to make object-oriented systems hard to maintain. In this study, we investigate if classes with code smells are more change-prone than classes without smells. Specifically, we test the general hypothesis: classes with code smells are not more change prone than other classes. We detect 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse, and study the relation between classes with these code smells and class change-proneness. We show that, in almost all releases of Azureus and Eclipse, classes with code smells are more change-prone than others, and that specific smells are more correlated than others to change-proneness. These results justify *a posteriori* previous work on the specification and detection of code smells and could help focusing quality assurance and testing activities.**

***Keywords*—Code Smells, Mining Software Repositories, Empirical Software Engineering.**

I. Context and Problem

In theory, code smells [1] are poor implementation choices, opposite to idioms [2] and, to some extent, to design patterns [3], in the sense that they pertain to implementation while design patterns pertain to the design. They are "poor" solutions to recurring implementation problems. In practice, code smells are in-between design and implementation: they may concern the design of a class, but they concretely manifest themselves in the source code as classes with specific implementation. They are usually revealed through particular metric values [4].

One example of a code smell is the ComplexClassOnly smell, which occurs in classes with a very high McCabe complexity when compared to other class in a system. At a higher level of abstraction, the presence of some specific code smells can, in turn, manifest in antipatterns [5], of which code smells are parts of. Studying the effects of

no previous work has contrasted the change-proneness of classes with code smells with this of other classes to study empirically the impact of code smells on this aspect of software evolution.

Goal. We want to investigate the relations between code smells and changes: First, we study whether classes with code smells have an increased likelihood of changing than other classes. Second, we study whether classes with more smells than others are more change-prone. Third, we study the relation between particular smells and change-proneness.

Contribution. We present an exploratory study investigating the relations between 29 code smells and changes occurring to classes in 9 releases of Azureus and 13 releases of Eclipse. We show that code smells *do* have a negative impact on classes, that certain kinds of smells *do* impact classes more than others, and that classes with more smells exhibit higher change-proneness.

Relevance. Understanding if code smells increase the risk of classes to change is important from the points of view of both researchers and practitioners.

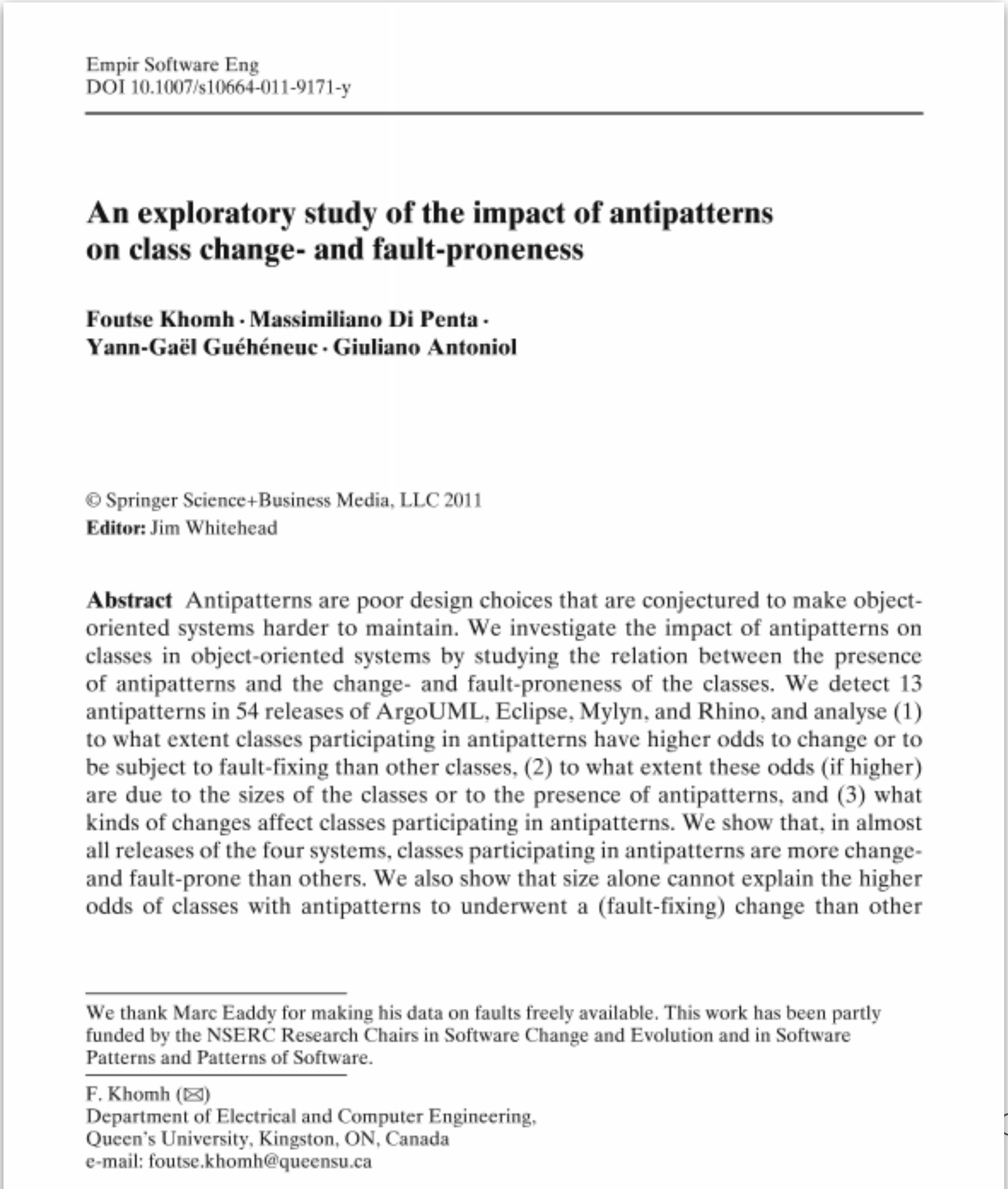
We bring evidence to researchers that (1) code smells *do* increase the number of changes that classes undergo, (2) the more smells a class has, the more change-prone it is, and (3) certain smells lead to more change-proneness than others. Therefore, this study justifies *a posteriori* previous work on code smells: within the limits of the threats to its validity, classes with code smells are more change-prone than others and therefore smells may indeed hinder software evolution; we empirically support such a conjecture reported in the literature [1], [6], [7], which is the premise of this study.

We also provide evidence to practitioners—developers, quality assurance personnel, and managers—of the impor-

Classes com cheiros de código tendem a ser mais alteradas

[Khomh et al., 2009]

Pesquisas relacionadas a cheiros de código



Empir Software Eng
DOI 10.1007/s10664-011-9171-y

An exploratory study of the impact of antipatterns on class change- and fault-proneness

Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

© Springer Science+Business Media, LLC 2011
Editor: Jim Whitehead

Abstract Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain. We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes. We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1) to what extent classes participating in antipatterns have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns. We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other

We thank Marc Eaddy for making his data on faults freely available. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

F. Khomh (✉)
Department of Electrical and Computer Engineering,
Queen's University, Kingston, ON, Canada
e-mail: foutse.khomh@queensu.ca

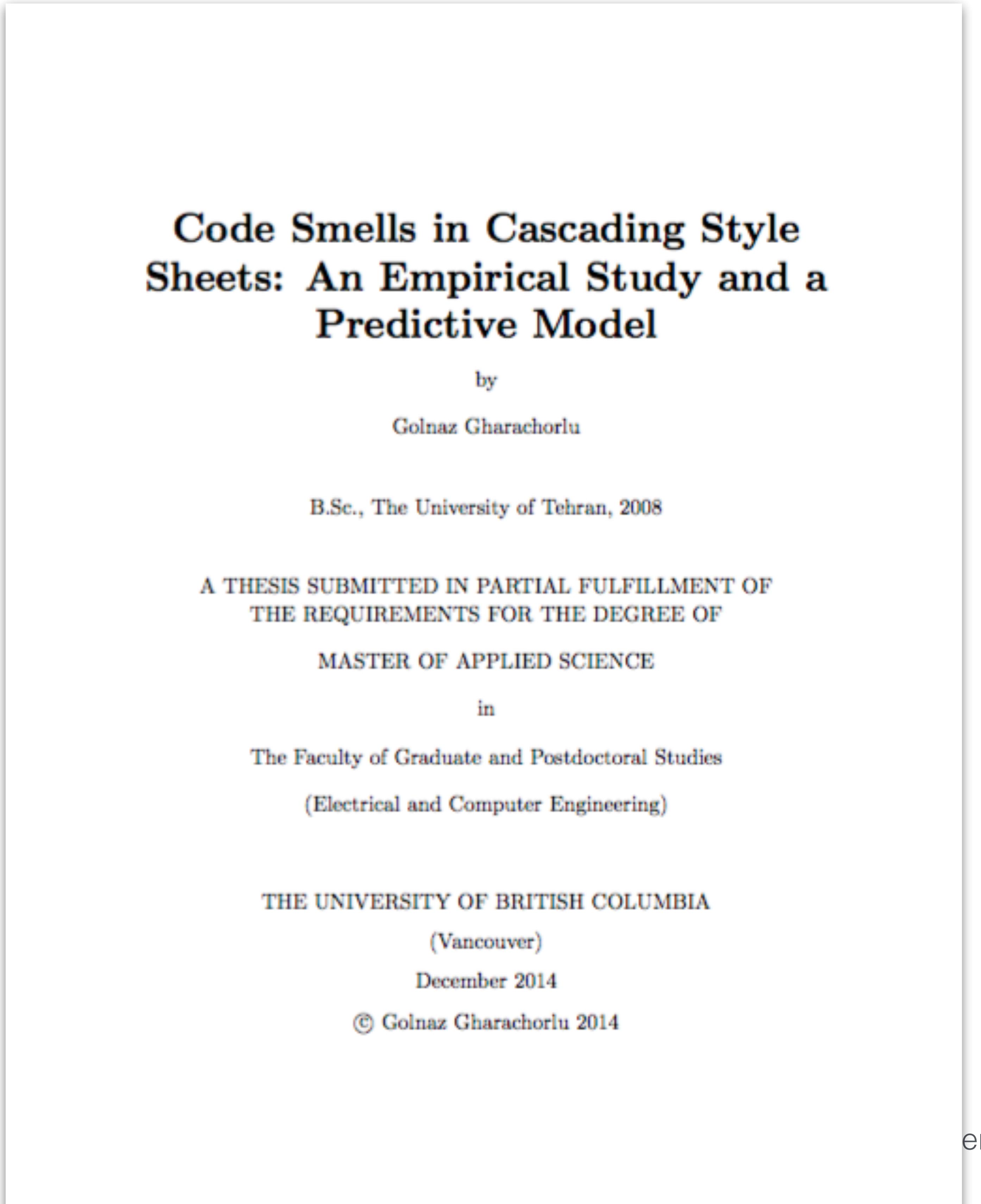
Classes com antipatterns tendem a ter mais alterações e falhas.

[Khomh et al., 2011]

Pesquisas relacionadas à plataforma Android

- Adrienne et al. [32] aborda sobre a efetividade do sistema de permissões.
- Diversas pesquisas [25, 43, 27, 28, 81, 87, 88] exploram vulnerabilidades do Android.
- Chin et al. [25], Kavitha et al. [43] abordam as vantagens e riscos envolvendo o sistema de troca de mensagens entre aplicativos e o sistema de permissões.
- Hu e Neamtiu [40] e Amalfitano et al. [17] abordam o tema testes de interface Android.
- Pesquisas em torno de autenticação segura através da identificação do dispositivo do usuário [29, 30, 82, 85]

Cheiros de códigos específicos à tecnologia ou plataforma



Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model

by

Golnaz Gharachorlu

B.Sc., The University of Tehran, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

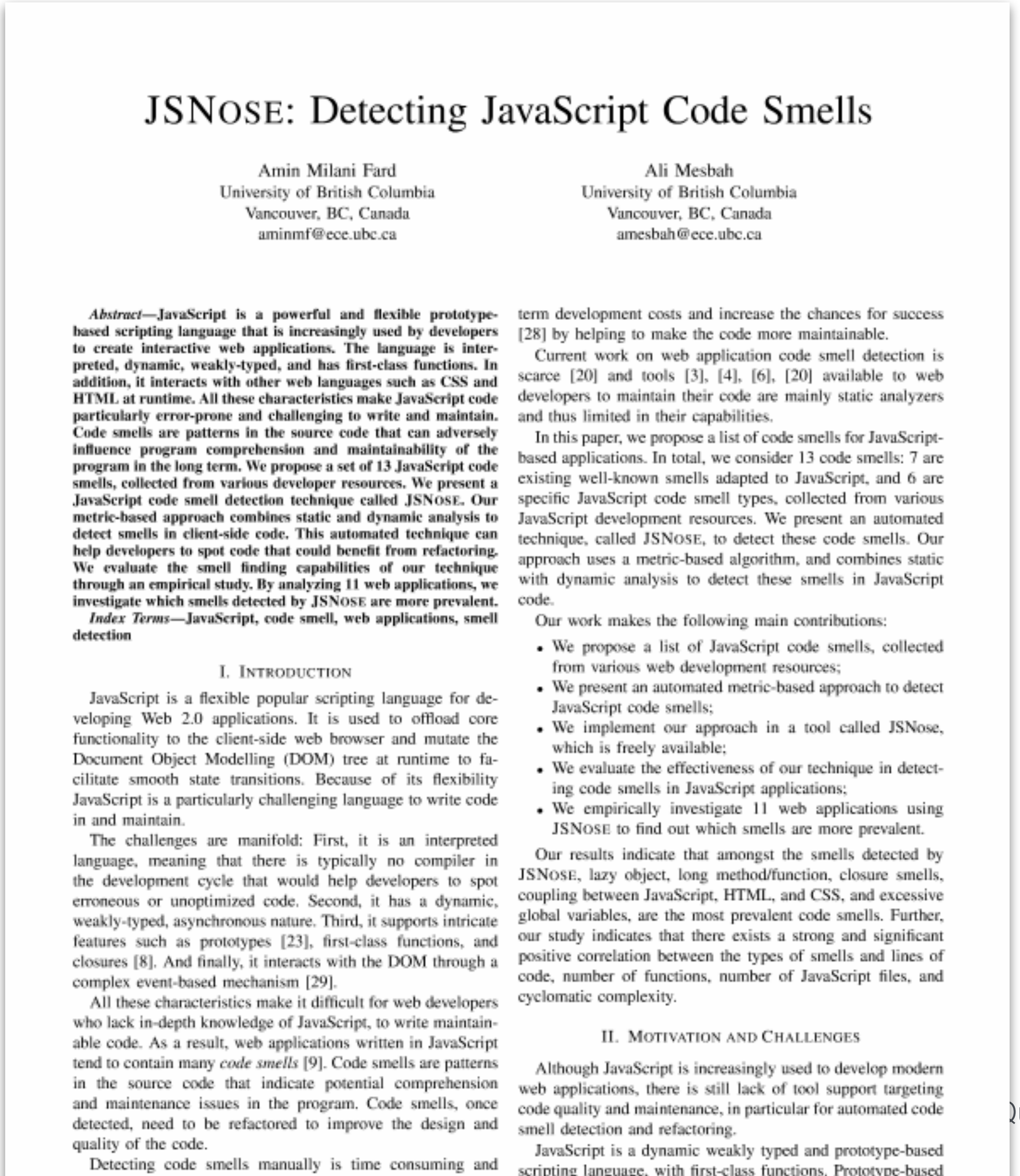
December 2014

© Golnaz Gharachorlu 2014

8 cheiros de código CSS

[Gharachorlu, 2008]

Cheiros de códigos específicos à tecnologia ou plataforma



JSNose: Detecting JavaScript Code Smells

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—JavaScript is a powerful and flexible prototype-based scripting language that is increasingly used by developers to create interactive web applications. The language is interpreted, dynamic, weakly-typed, and has first-class functions. In addition, it interacts with other web languages such as CSS and HTML at runtime. All these characteristics make JavaScript code particularly error-prone and challenging to write and maintain. Code smells are patterns in the source code that can adversely influence program comprehension and maintainability of the program in the long term. We propose a set of 13 JavaScript code smells, collected from various developer resources. We present a JavaScript code smell detection technique called JSNose. Our metric-based approach combines static and dynamic analysis to detect smells in client-side code. This automated technique can help developers to spot code that could benefit from refactoring. We evaluate the smell finding capabilities of our technique through an empirical study. By analyzing 11 web applications, we investigate which smells detected by JSNose are more prevalent.**

***Index Terms*—JavaScript, code smell, web applications, smell detection**

I. Introduction

JavaScript is a flexible popular scripting language for developing Web 2.0 applications. It is used to offload core functionality to the client-side web browser and mutate the Document Object Modelling (DOM) tree at runtime to facilitate smooth state transitions. Because of its flexibility JavaScript is a particularly challenging language to write code in and maintain.

The challenges are manifold: First, it is an interpreted language, meaning that there is typically no compiler in the development cycle that would help developers to spot erroneous or unoptimized code. Second, it has a dynamic, weakly-typed, asynchronous nature. Third, it supports intricate features such as prototypes [23], first-class functions, and closures [8]. And finally, it interacts with the DOM through a complex event-based mechanism [29].

All these characteristics make it difficult for web developers who lack in-depth knowledge of JavaScript, to write maintainable code. As a result, web applications written in JavaScript tend to contain many *code smells* [9]. Code smells are patterns in the source code that indicate potential comprehension and maintenance issues in the program. Code smells, once detected, need to be refactored to improve the design and quality of the code.

Detecting code smells manually is time consuming and term development costs and increase the chances for success [28] by helping to make the code more maintainable.

Current work on web application code smell detection is scarce [20] and tools [3], [4], [6], [20] available to web developers to maintain their code are mainly static analyzers and thus limited in their capabilities.

In this paper, we propose a list of code smells for JavaScript-based applications. In total, we consider 13 code smells: 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smell types, collected from various JavaScript development resources. We present an automated technique, called JSNose, to detect these code smells. Our approach uses a metric-based algorithm, and combines static with dynamic analysis to detect these smells in JavaScript code.

Our work makes the following main contributions:

- We propose a list of JavaScript code smells, collected from various web development resources;
- We present an automated metric-based approach to detect JavaScript code smells;
- We implement our approach in a tool called JSNose, which is freely available;
- We evaluate the effectiveness of our technique in detecting code smells in JavaScript applications;
- We empirically investigate 11 web applications using JSNose to find out which smells are more prevalent.

Our results indicate that amongst the smells detected by JSNose, lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables, are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and lines of code, number of functions, number of JavaScript files, and cyclomatic complexity.

II. Motivation and Challenges

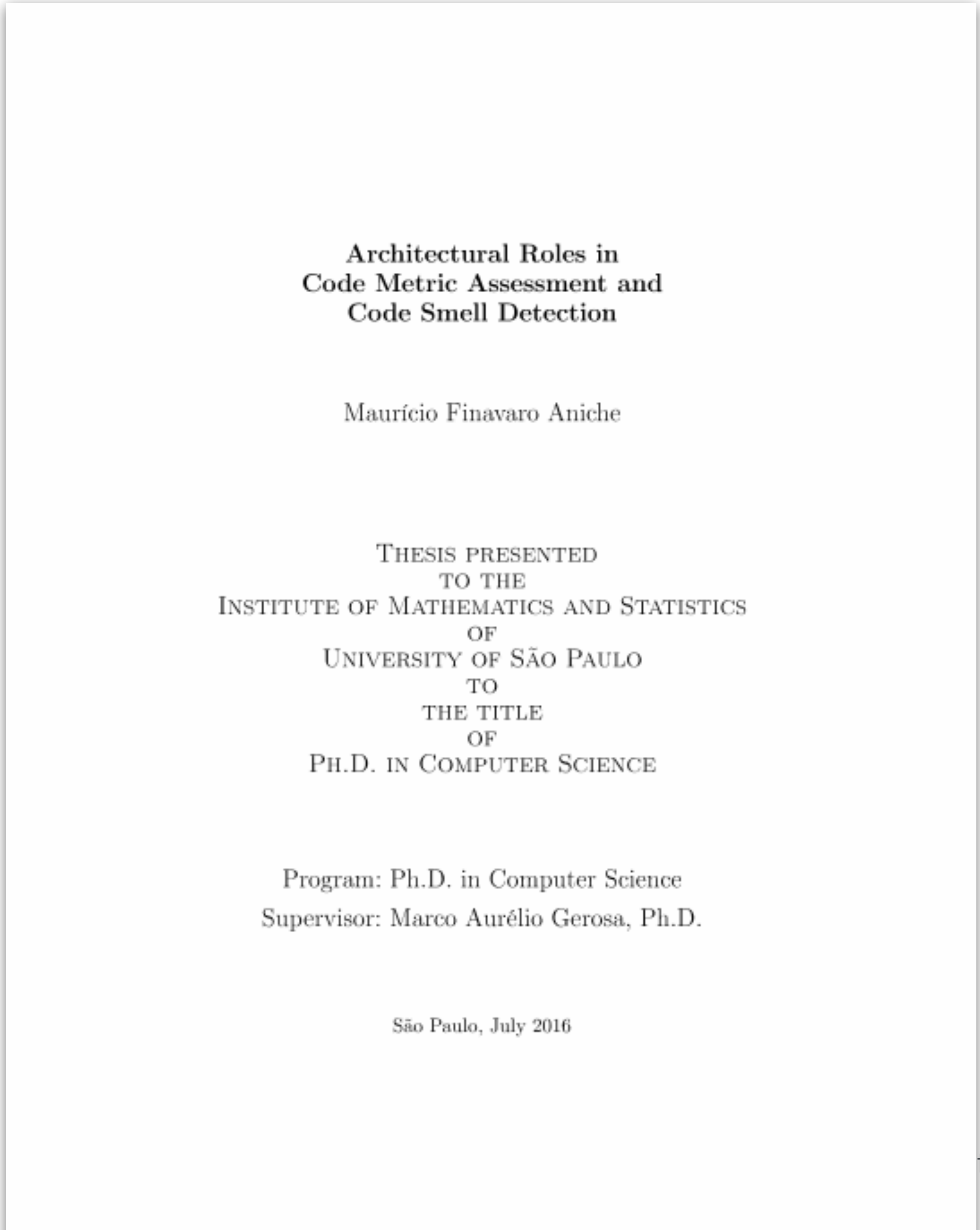
Although JavaScript is increasingly used to develop modern web applications, there is still lack of tool support targeting code quality and maintenance, in particular for automated code smell detection and refactoring.

JavaScript is a dynamic weakly typed and prototype-based scripting language, with first-class functions. Prototype-based

6 cheiros de código Javascript

[Fard e Ali, 2013]

Cheiros de códigos específicos à tecnologia ou plataforma



Architectural Roles in Code Metric Assessment and Code Smell Detection

Maurício Finavaro Aniche

THESIS PRESENTED TO THE INSTITUTE OF MATHEMATICS AND STATISTICS OF UNIVERSITY OF SÃO PAULO TO THE TITLE OF PH.D. IN COMPUTER SCIENCE

Program: Ph.D. in Computer Science
Supervisor: Marco Aurélio Gerosa, Ph.D.

São Paulo, July 2016

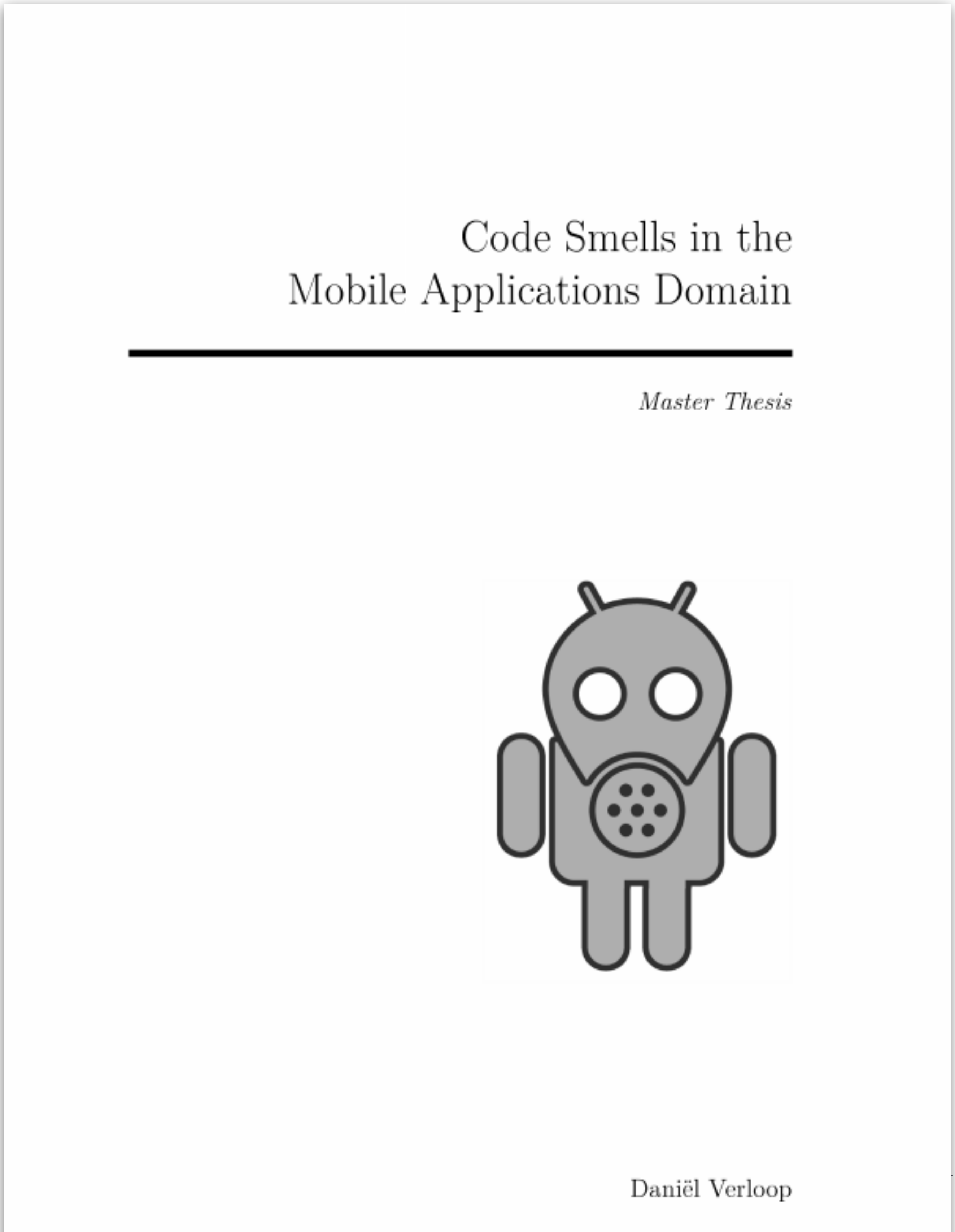
6 cheiros de código Spring MVC

[Aniche et al., 2016]

Métodos usado para definição dos cheiros de código específicos

Resultado da Pesquisa	Método de Coleta de Dados	Método de Definição dos Cheiros de Códigos
6 cheiros de código específicos ao framework Spring MVC [18].	Questionários online e entrevistas. Total de 53 desenvolvedores.	• Categorização individual por dois dos autores. • Definição final sob consenso dos dois autores. • Validação com autoridade em Spring MVC.
6 cheiros de código específicos ao Javascript [31].	Blog e sites técnicos sobre Javascript e livros sobre Javascript.	Foi realizado um estudo sobre dados de sites, blogs e livros sobre Javascript.
8 cheiros de código específicos de CSS [37].	500 sites com um total de 5.060 arquivos CSS sumarizando 10 milhões de linhas de código CSS.	Foi realizada uma ampla investigação em sites técnicos sobre CSS e ferramentas de análise de qualidade de código CSS.

Cheiros de código relacionados à plataforma Android



Code Smells in the
Mobile Applications Domain

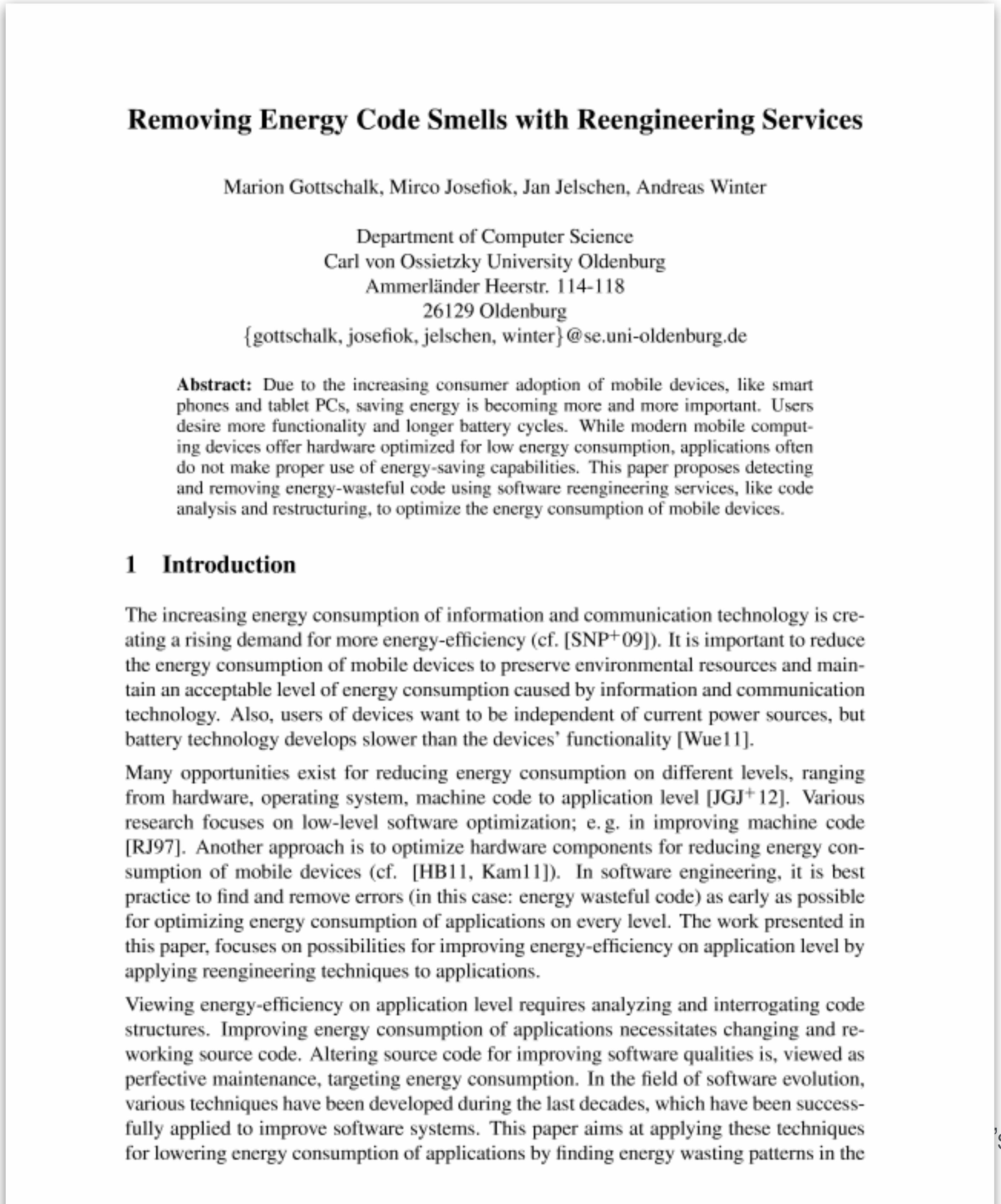
Master Thesis

Daniël Verloop

Cheiros de código tradicionais em projetos Android

Analisa a probabilidade de cheiros de código tradicionais em classes Android. [Verloop, 2013]

Cheiros de código relacionados à plataforma Android



Removing Energy Code Smells with Reengineering Services

Marion Gottschalk, Mirco Josefiok, Jan Jelschen, Andreas Winter

Department of Computer Science
Carl von Ossietzky University Oldenburg
Ammerländer Heerstr. 114-118
26129 Oldenburg
{gottschalk, josefiok, jelschen, winter}@se.uni-oldenburg.de

Abstract: Due to the increasing consumer adoption of mobile devices, like smart phones and tablet PCs, saving energy is becoming more and more important. Users desire more functionality and longer battery cycles. While modern mobile computing devices offer hardware optimized for low energy consumption, applications often do not make proper use of energy-saving capabilities. This paper proposes detecting and removing energy-wasteful code using software reengineering services, like code analysis and restructuring, to optimize the energy consumption of mobile devices.

1 Introduction

The increasing energy consumption of information and communication technology is creating a rising demand for more energy-efficiency (cf. [SNP$^+$09]). It is important to reduce the energy consumption of mobile devices to preserve environmental resources and maintain an acceptable level of energy consumption caused by information and communication technology. Also, users of devices want to be independent of current power sources, but battery technology develops slower than the devices' functionality [Wue11].

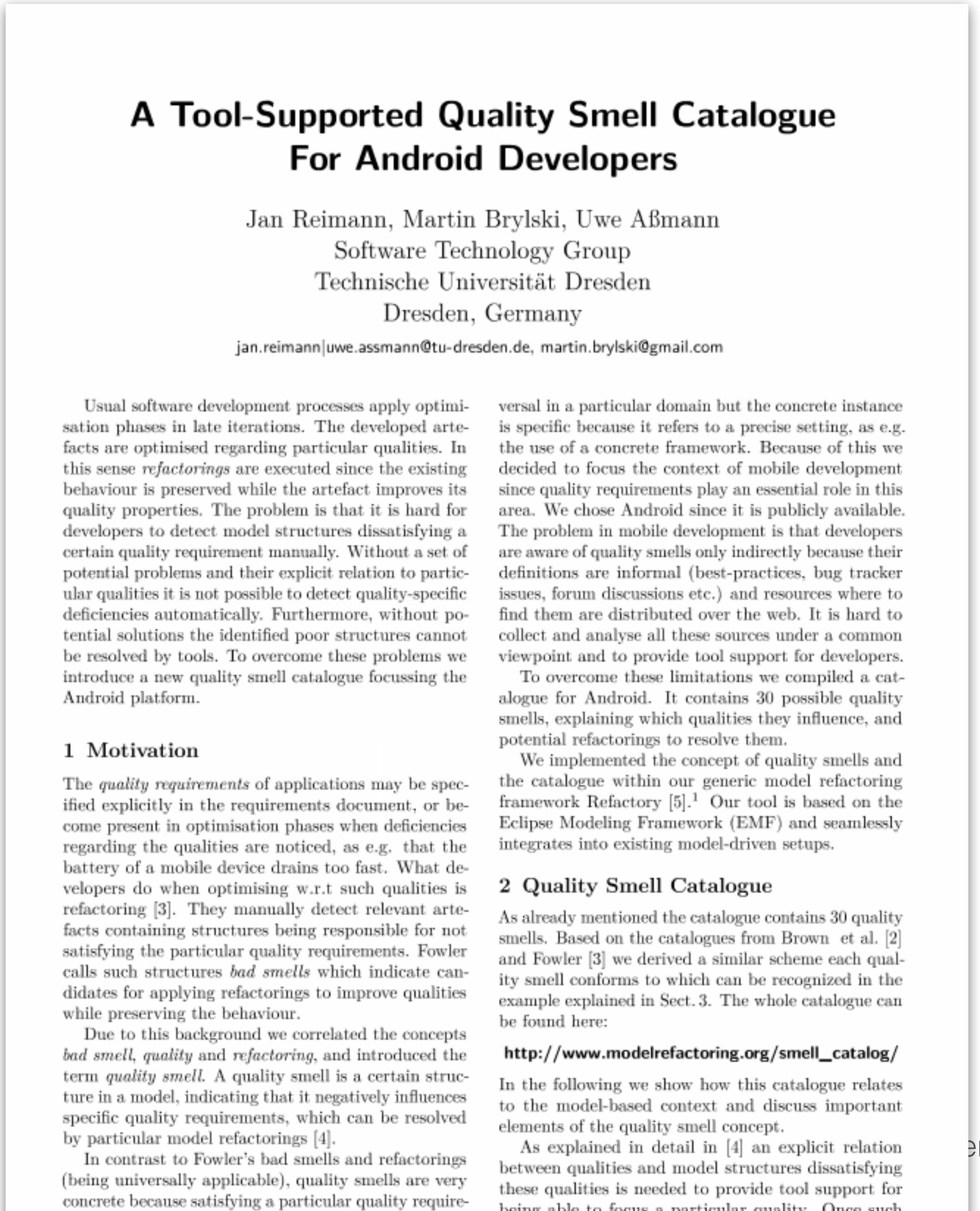
Many opportunities exist for reducing energy consumption on different levels, ranging from hardware, operating system, machine code to application level [JGJ$^+$12]. Various research focuses on low-level software optimization; e. g. in improving machine code [RJ97]. Another approach is to optimize hardware components for reducing energy consumption of mobile devices (cf. [HB11, Kam11]). In software engineering, it is best practice to find and remove errors (in this case: energy wasteful code) as early as possible for optimizing energy consumption of applications on every level. The work presented in this paper, focuses on possibilities for improving energy-efficiency on application level by applying reengineering techniques to applications.

Viewing energy-efficiency on application level requires analyzing and interrogating code structures. Improving energy consumption of applications necessitates changing and reworking source code. Altering source code for improving software qualities is, viewed as perfective maintenance, targeting energy consumption. In the field of software evolution, various techniques have been developed during the last decades, which have been successfully applied to improve software systems. This paper aims at applying these techniques for lowering energy consumption of applications by finding energy wasting patterns in the

Cheiros de código Android sobre energia

Produzem um catálogo com 6 cheiros de códigos para a redução de consumo de energia. [Gottschalk e Jelschen, 2013]

Cheiros de código relacionados à plataforma Android



A Tool-Supported Quality Smell Catalogue For Android Developers

Jan Reimann, Martin Brylski, Uwe Aßmann
Software Technology Group
Technische Universität Dresden
Dresden, Germany
jan.reimann|uwe.assmann@tu-dresden.de, martin.brylski@gmail.com

Usual software development processes apply optimisation phases in late iterations. The developed artefacts are optimised regarding particular qualities. In this sense *refactorings* are executed since the existing behaviour is preserved while the artefact improves its quality properties. The problem is that it is hard for developers to detect model structures dissatisfying a certain quality requirement manually. Without a set of potential problems and their explicit relation to particular qualities it is not possible to detect quality-specific deficiencies automatically. Furthermore, without potential solutions the identified poor structures cannot be resolved by tools. To overcome these problems we introduce a new quality smell catalogue focussing the Android platform.

1 Motivation

The *quality requirements* of applications may be specified explicitly in the requirements document, or become present in optimisation phases when deficiencies regarding the qualities are noticed, as e.g. that the battery of a mobile device drains too fast. What developers do when optimising w.r.t such qualities is refactoring [3]. They manually detect relevant artefacts containing structures being responsible for not satisfying the particular quality requirements. Fowler calls such structures *bad smells* which indicate candidates for applying refactorings to improve qualities while preserving the behaviour.

Due to this background we correlated the concepts *bad smell*, *quality* and *refactoring*, and introduced the term *quality smell*. A quality smell is a certain structure in a model, indicating that it negatively influences specific quality requirements, which can be resolved by particular model refactorings [4].

In contrast to Fowler's bad smells and refactorings (being universally applicable), quality smells are very concrete because satisfying a particular quality requirement is not universal in a particular domain but the concrete instance is specific because it refers to a precise setting, as e.g. the use of a concrete framework. Because of this we decided to focus the context of mobile development since quality requirements play an essential role in this area. We chose Android since it is publicly available. The problem in mobile development is that developers are aware of quality smells only indirectly because their definitions are informal (best-practices, bug tracker issues, forum discussions etc.) and resources where to find them are distributed over the web. It is hard to collect and analyse all these sources under a common viewpoint and to provide tool support for developers.

To overcome these limitations we compiled a catalogue for Android. It contains 30 possible quality smells, explaining which qualities they influence, and potential refactorings to resolve them.

We implemented the concept of quality smells and the catalogue within our generic model refactoring framework Refactory [5].[1] Our tool is based on the Eclipse Modeling Framework (EMF) and seamlessly integrates into existing model-driven setups.

2 Quality Smell Catalogue

As already mentioned the catalogue contains 30 quality smells. Based on the catalogues from Brown et al. [2] and Fowler [3] we derived a similar scheme each quality smell conforms to which can be recognized in the example explained in Sect. 3. The whole catalogue can be found here:

http://www.modelrefactoring.org/smell_catalog/

In the following we show how this catalogue relates to the model-based context and discuss important elements of the quality smell concept.

As explained in detail in [4] an explicit relation between qualities and model structures dissatisfying these qualities is needed to provide tool support for

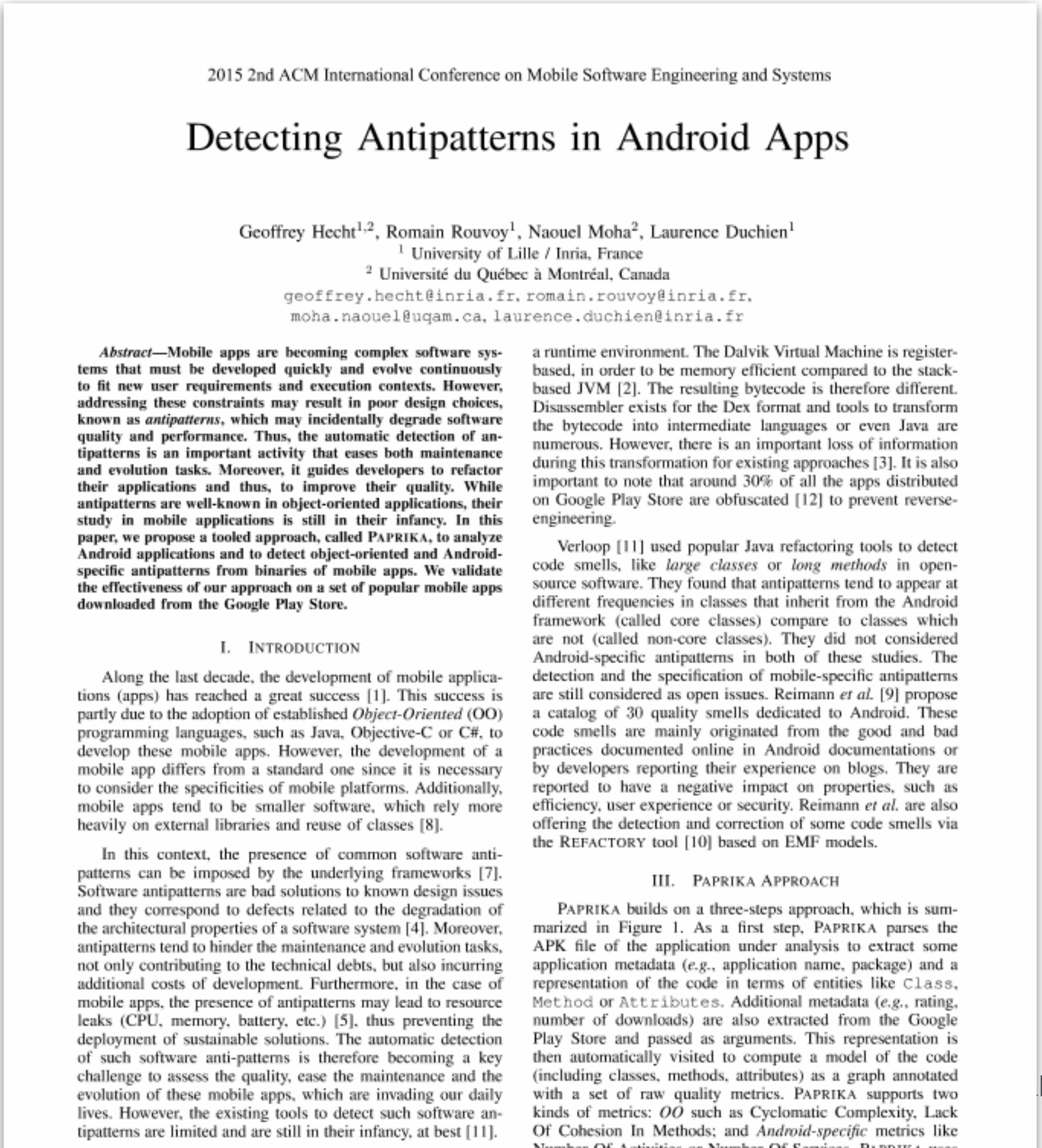
Cheiros de código de qualidade

Catálogo com 30 cheiros de qualidade sobre requisitos do usuário, consumo inteligente de recursos e segurança.

[Reimann et al., 2013]

Catálogo disponível online em www.modelrefactoring.org/smell_catalog

Cheiros de código relacionados à plataforma Android



2015 2nd ACM International Conference on Mobile Software Engineering and Systems

Detecting Antipatterns in Android Apps

Geoffrey Hecht[1,2], Romain Rouvoy[1], Naouel Moha[2], Laurence Duchien[1]
[1] University of Lille / Inria, France
[2] Université du Québec à Montréal, Canada
geoffrey.hecht@inria.fr, romain.rouvoy@inria.fr,
moha.naouel@uqam.ca, laurence.duchien@inria.fr

***Abstract*—Mobile apps are becoming complex software systems that must be developed quickly and evolve continuously to fit new user requirements and execution contexts. However, addressing these constraints may result in poor design choices, known as *antipatterns*, which may incidentally degrade software quality and performance. Thus, the automatic detection of antipatterns is an important activity that eases both maintenance and evolution tasks. Moreover, it guides developers to refactor their applications and thus, to improve their quality. While antipatterns are well-known in object-oriented applications, their study in mobile applications is still in their infancy. In this paper, we propose a tooled approach, called PAPRIKA, to analyze Android applications and to detect object-oriented and Android-specific antipatterns from binaries of mobile apps. We validate the effectiveness of our approach on a set of popular mobile apps downloaded from the Google Play Store.**

I. INTRODUCTION

Along the last decade, the development of mobile applications (apps) has reached a great success [1]. This success is partly due to the adoption of established *Object-Oriented* (OO) programming languages, such as Java, Objective-C or C#, to develop these mobile apps. However, the development of a mobile app differs from a standard one since it is necessary to consider the specificities of mobile platforms. Additionally, mobile apps tend to be smaller software, which rely more heavily on external libraries and reuse of classes [8].

In this context, the presence of common software antipatterns can be imposed by the underlying frameworks [7]. Software antipatterns are bad solutions to known design issues and they correspond to defects related to the degradation of the architectural properties of a software system [4]. Moreover, antipatterns tend to hinder the maintenance and evolution tasks, not only contributing to the technical debts, but also incurring additional costs of development. Furthermore, in the case of mobile apps, the presence of antipatterns may lead to resource leaks (CPU, memory, battery, etc.) [5], thus preventing the deployment of sustainable solutions. The automatic detection of such software anti-patterns is therefore becoming a key challenge to assess the quality, ease the maintenance and the evolution of these mobile apps, which are invading our daily lives. However, the existing tools to detect such software antipatterns are limited and are still in their infancy, at best [11].

a runtime environment. The Dalvik Virtual Machine is register-based, in order to be memory efficient compared to the stack-based JVM [2]. The resulting bytecode is therefore different. Disassembler exists for the Dex format and tools to transform the bytecode into intermediate languages or even Java are numerous. However, there is an important loss of information during this transformation for existing approaches [3]. It is also important to note that around 30% of all the apps distributed on Google Play Store are obfuscated [12] to prevent reverse-engineering.

Verloop [11] used popular Java refactoring tools to detect code smells, like *large classes* or *long methods* in open-source software. They found that antipatterns tend to appear at different frequencies in classes that inherit from the Android framework (called core classes) compare to classes which are not (called non-core classes). They did not considered Android-specific antipatterns in both of these studies. The detection and the specification of mobile-specific antipatterns are still considered as open issues. Reimann *et al.* [9] propose a catalog of 30 quality smells dedicated to Android. These code smells are mainly originated from the good and bad practices documented online in Android documentations or by developers reporting their experience on blogs. They are reported to have a negative impact on properties, such as efficiency, user experience or security. Reimann *et al.* are also offering the detection and correction of some code smells via the REFACTORY tool [10] based on EMF models.

III. PAPRIKA APPROACH

PAPRIKA builds on a three-steps approach, which is summarized in Figure 1. As a first step, PAPRIKA parses the APK file of the application under analysis to extract some application metadata (*e.g.*, application name, package) and a representation of the code in terms of entities like `Class`, `Method` or `Attributes`. Additional metadata (*e.g.*, rating, number of downloads) are also extracted from the Google Play Store and passed as arguments. This representation is then automatically visited to compute a model of the code (including classes, methods, attributes) as a graph annotated with a set of raw quality metrics. PAPRIKA supports two kinds of metrics: *OO* such as Cyclomatic Complexity, Lack Of Cohesion In Methods; and *Android-specific* metrics like

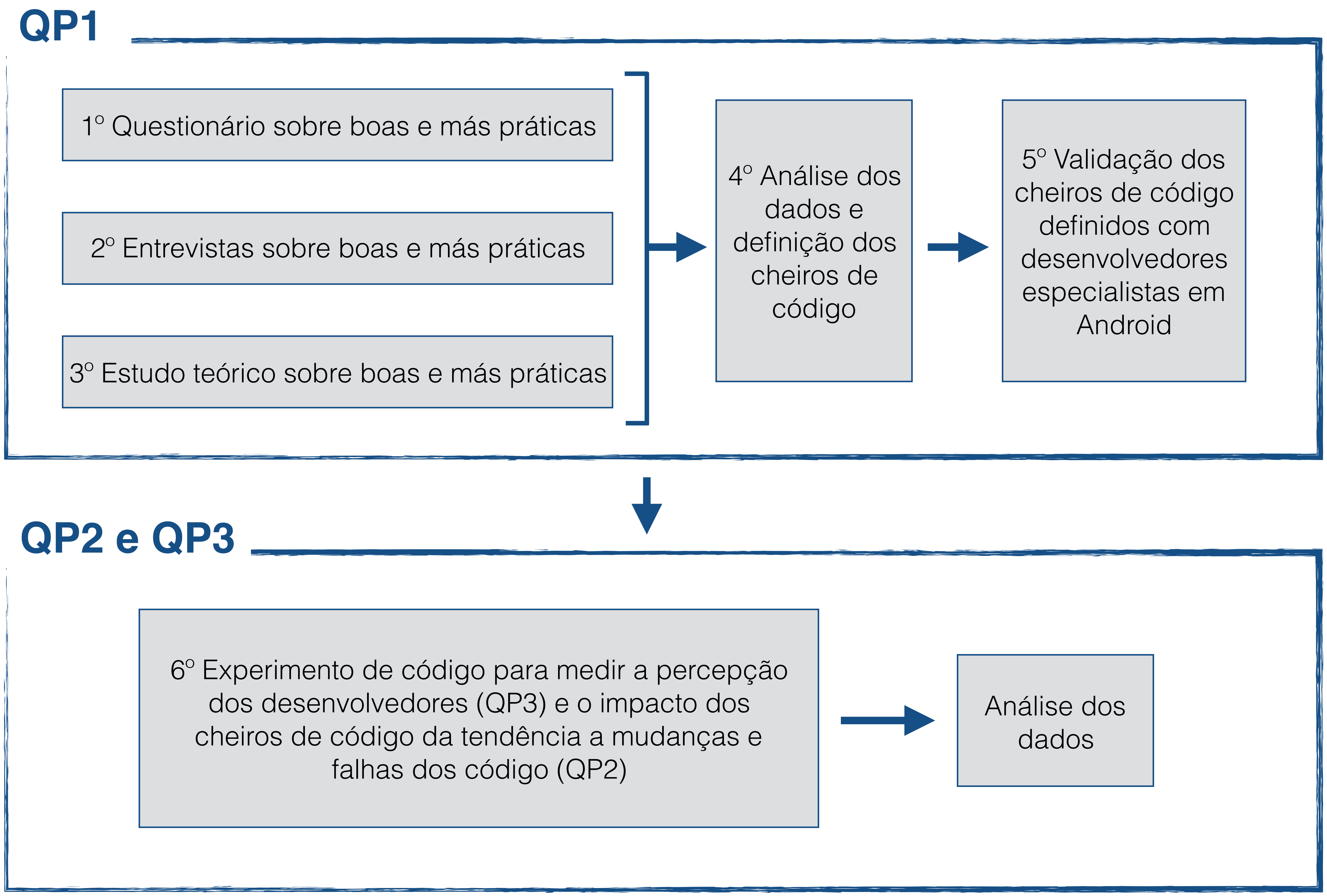
Cheiros de código tradicionais e específicos em projetos Android

Utiliza a ferramenta <u>PAPRIKA</u> para identificar em projetos Android cheiros de código tradicionais e específicos [60].
[Hecht et al., 2013]

Métodos usado para definição dos cheiros de código

Resultado da Pesquisa	Método de Coleta de Dados	Método de Definição dos Cheiros de Códigos
6 cheiros de código Android sobre energia [38].	Literatura.	Pesquisando literatura, um conjunto inicial de cheiros de código de energia foi identificado.
30 cheiros de código Android sobre qualidade [60].	Blog e sites técnicos sobre Android e documentação oficial do Android.	Não especifica.

Design da pesquisa

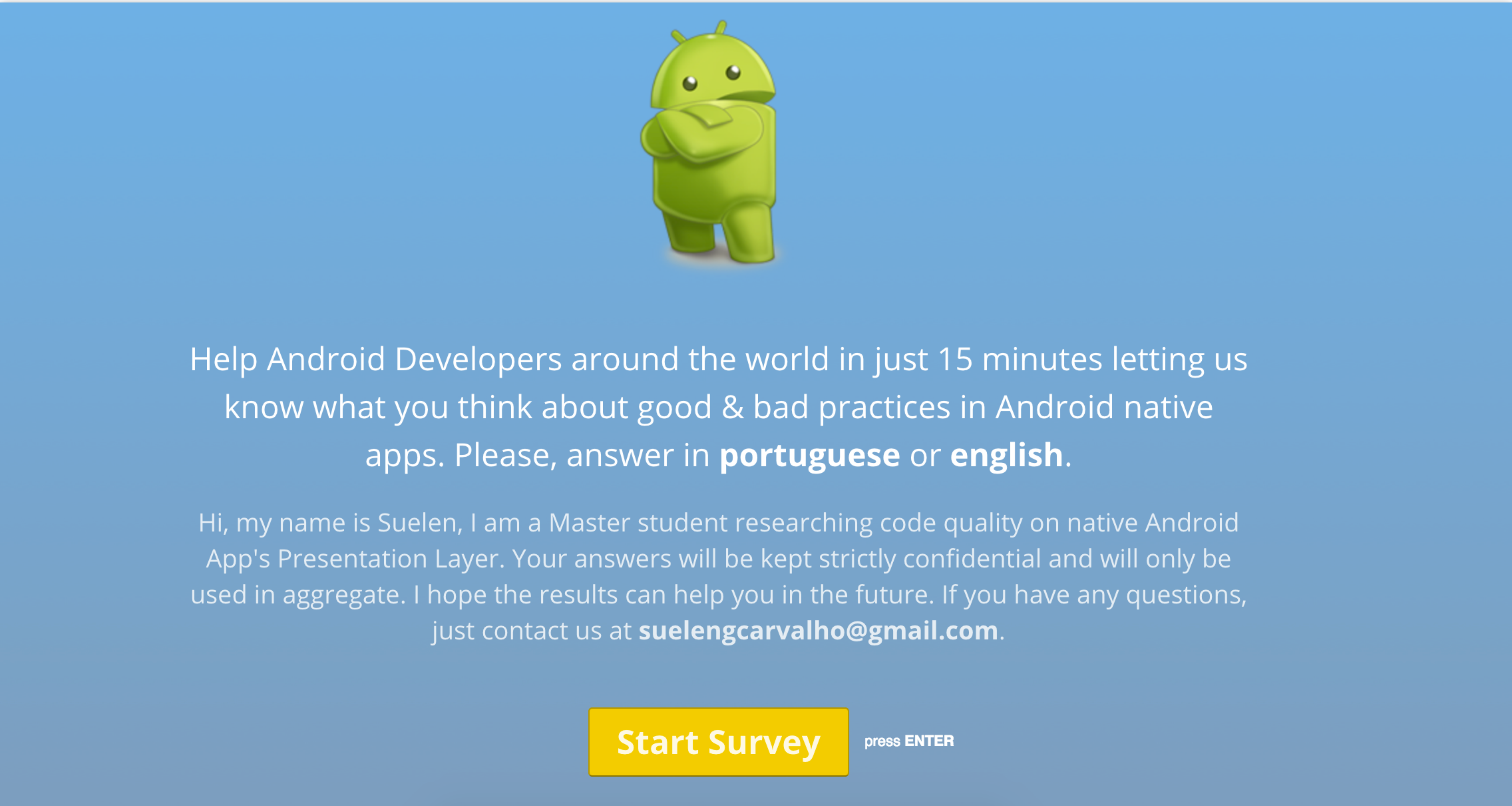

QP1
1º Questionário sobre boas e más práticas
2º Entrevistas sobre boas e más práticas
3º Estudo teórico sobre boas e más práticas
4º Análise dos dados e definição dos cheiros de código
5º Validação dos cheiros de código definidos com desenvolvedores especialistas em Android
QP2 e QP3
6º Experimento de código para medir a percepção dos desenvolvedores (QP3) e o impacto dos cheiros de código da tendência a mudanças e falhas dos código (QP2)
Análise dos dados

1º Questionário sobre boas e más práticas

- Características
  - Questionário online sobre boas e más praticas em elementos da camada de apresentação Android.
  - Aberto a comunidade de desenvolvedores do Brazil e exterior.
  - Divulgado em redes sociais como LinkedIn, Twitter, Facebook e grupos de desenvolvedores Android.
  - Questionário em inglês.
- Resultados preliminares:
  - Total de **44 respostas** vindas de mais de 5 países diferentes.
  - **88%** possuem 3+ anos de experiência com desenvolvimento de software e **71%** possuem 3+ anos de experiência com Android.
  - **86%** se consideram proficientes em Java e **89%** se consideram proficientes em Android.

1º Questionário sobre boas e más práticas (cont.)

tinyurl.com/android-good-and-bad-practices



Help Android Developers around the world in just 15 minutes letting us know what you think about good & bad practices in Android native apps. Please, answer in **portuguese** or **english**.

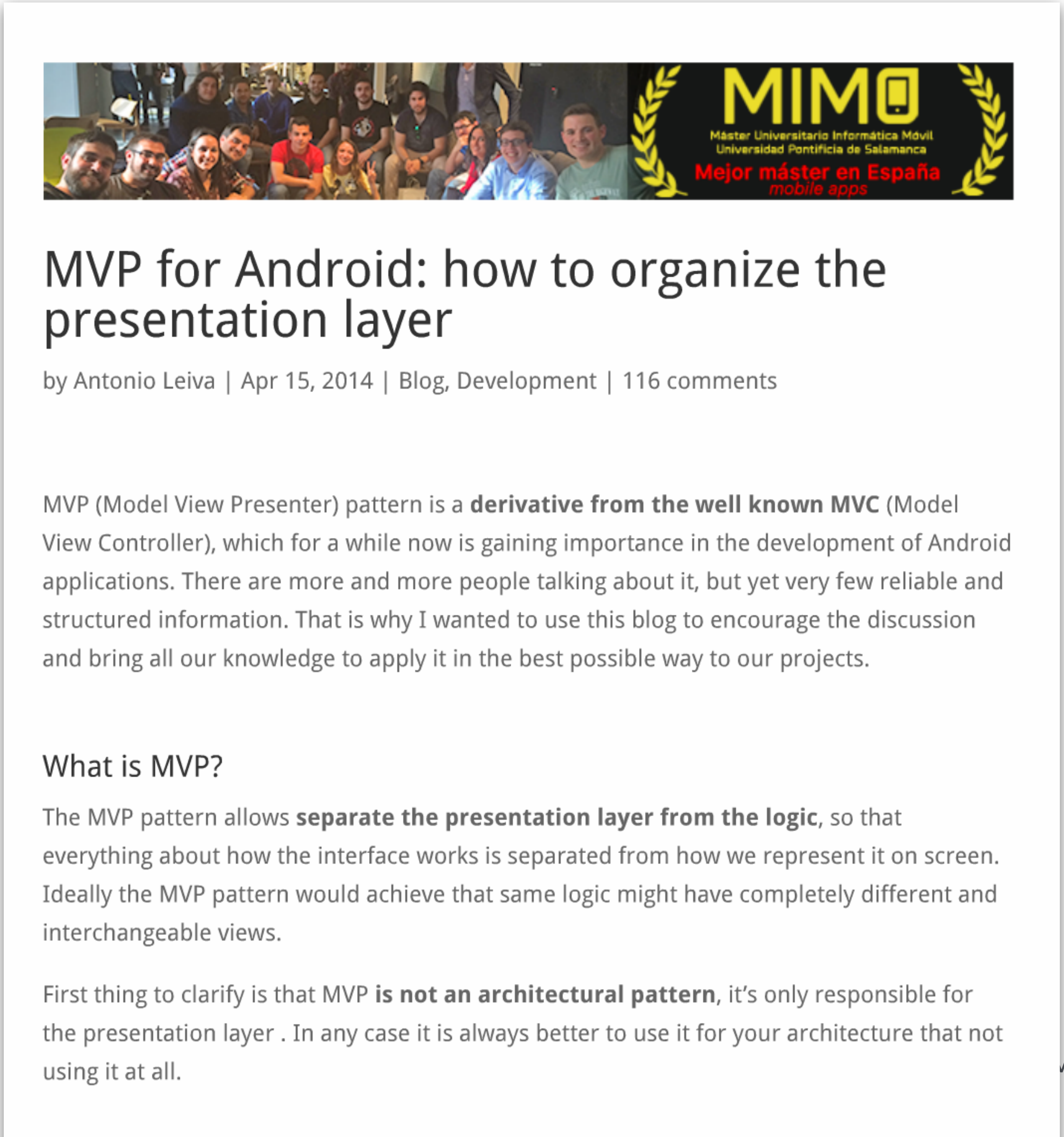
Hi, my name is Suelen, I am a Master student researching code quality on native Android App's Presentation Layer. Your answers will be kept strictly confidential and will only be used in aggregate. I hope the results can help you in the future. If you have any questions, just contact us at **suelengcarvalho@gmail.com**.

2º Entrevistas sobre boas e más práticas

- Características
  - Entrevistas não-estruturadas com perguntas abertas por pautas.
  - As pautas serão baseadas no questionário online.
- Participantes
  - Desenvolvedores Android convidados do Brazil e exterior.
  - Os áudios serão armazenados e transcritos.

3º Estudo teórico sobre boas e más práticas

- Buscas por textos relacionados a boas e más práticas na camada de apresentação Android em sites, blogs e fóruns sobre Android.


MIMO
Máster Universitario Informática Móvil
Universidad Pontificia de Salamanca
Mejor máster en España
mobile apps

MVP for Android: how to organize the presentation layer

by Antonio Leiva | Apr 15, 2014 | Blog, Development | 116 comments

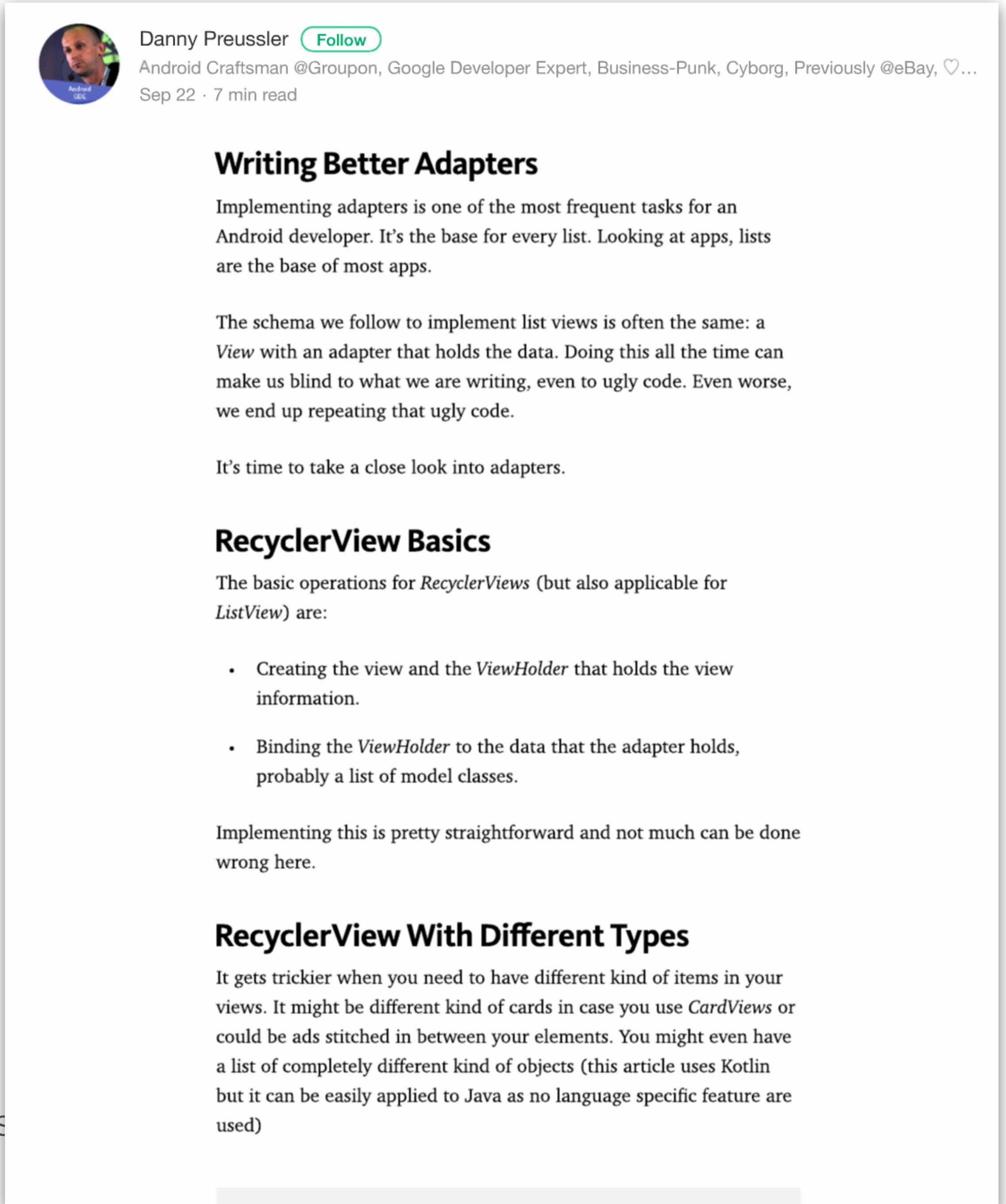
MVP (Model View Presenter) pattern is a **derivative from the well known MVC** (Model View Controller), which for a while now is gaining importance in the development of Android applications. There are more and more people talking about it, but yet very few reliable and structured information. That is why I wanted to use this blog to encourage the discussion and bring all our knowledge to apply it in the best possible way to our projects.

What is MVP?

The MVP pattern allows **separate the presentation layer from the logic**, so that everything about how the interface works is separated from how we represent it on screen. Ideally the MVP pattern would achieve that same logic might have completely different and interchangeable views.

First thing to clarify is that MVP **is not an architectural pattern**, it's only responsible for the presentation layer . In any case it is always better to use it for your architecture that not using it at all.

Master's



Danny Preussler Follow
Android Craftsman @Groupon, Google Developer Expert, Business-Punk, Cyborg, Previously @eBay, ♡...
Sep 22 · 7 min read

Writing Better Adapters

Implementing adapters is one of the most frequent tasks for an Android developer. It's the base for every list. Looking at apps, lists are the base of most apps.

The schema we follow to implement list views is often the same: a *View* with an adapter that holds the data. Doing this all the time can make us blind to what we are writing, even to ugly code. Even worse, we end up repeating that ugly code.

It's time to take a close look into adapters.

RecyclerView Basics

The basic operations for *RecyclerViews* (but also applicable for *ListView*) are:

- Creating the view and the *ViewHolder* that holds the view information.
- Binding the *ViewHolder* to the data that the adapter holds, probably a list of model classes.

Implementing this is pretty straightforward and not much can be done wrong here.

RecyclerView With Different Types

It gets trickier when you need to have different kind of items in your views. It might be different kind of cards in case you use *CardViews* or could be ads stitched in between your elements. You might even have a list of completely different kind of objects (this article uses Kotlin but it can be easily applied to Java as no language specific feature are used)

4º Análise e definição dos cheiros de código

- Categorização manual das boas e más práticas recuperadas por meio do questionário online, entrevista e buscas em sites, blogs e fóruns sobre Android.
- Definição dos cheiros de código com base nas boas e más práticas categorizadas.
- Para a escrita dos cheiros de código, pretende-se usar o formato similar aos cheiros de código definidor por Fowler [33].

5º Validação com desenvolvedores especialistas em Android

- Será conduzida uma entrevista não estruturada com desenvolvedor especialista/referência em Android com o objetivo de avaliar a relevância ou não dos cheiros de código definidos.
- Este desenvolvedor será convidado.
- O convidado deve ser especialista em Android.
  - Já temos uma pessoa em mente que é o Neto Marin, developer advocate de Android no Google Brasil.

6º Experimento sobre percepção e impacto na tendência a mudanças e falhas

- Pretende-se realizar um experimento de código onde:
  - será apresentado um projeto a desenvolvedores Android e será solicitado que implementem uma funcionalidade.
  - O projeto apresentado possuirá duas versões, uma com os cheiros de código e outra “limpa”. Cada desenvolvedor irá receber uma versão ou a outra, sem saber que há duas versões.
  - Cada desenvolvedor usará uma branch específica e será solicitado que faça commits ao longo do desenvolvimento da funcionalidade.
- Os dados para responder a QP2 serão recuperados do histórico de commits da branch.
- Ao final do experimento será solicitado ao desenvolvedor que preencha um questionário online sobre sua percepção sobre o código do projeto.

6º Experimento sobre percepção e impacto na tendência a mudanças e falhas

- Entretanto, ainda estuda-se aplicar a mesma metodologia usada na tese de doutorado no Aniche [18]
  - Avaliação de projetos existentes buscando pelos cheiros de código definidos e comparar com cheiros tradicionais.
  - Questionário online sobre percepção dos desenvolvedores com relação aos cheiros de código definidos.

Cronograma de atividades

	2016		2017			
Atividades	3º Tri	4º Tri	1º Tri	2º Tri	3º Tri	4º Tri
Bibliografia e Trabalhos Relacionados	●	●				
Survey Boas e Más práticas Android		●				
Entrevista Boas e Más práticas Android			●			
Derivação dos cheiros de código			●			
Validação cheiros de código com Especialista			●			
Experimento Impacto em Mudanças/Defeitos				●		
Experimento Percepção Desenvolvedores				●		
Escrita da Dissertação	●	●	●	●	●	●
Defesa						●

Perguntas? :)

Suelen Goularte Carvalho
Orientador: Prof. Dr. Marco Aurélio Gerosa

suelengc@ime.usp.br
www.suelengc.com