Recommending when Design Technical Debt Should be Self-Admitted

Fiorella Zampetti[1], Cedric Noiseux[2], Giuliano Antoniol[2], Foutse Khomh[2], Massimiliano Di Penta[1]

[1]University of Sannio, Italy

[2]Polytechnique Montréal, Canada

fiorella.zampetti@unisannio.it
@fio_zampetti

Technical Debt refers to

" not quite right code which we postpone making it right."

[Ward Cunningham]

Technical Debt introduction is…

intentional

in · ten · tion · al

Adjective: Done on purpose; deliberate.

FOCUS: TECHNICAL DEBT

A Balancing Act: What Software Practitioners Have to Say about Technical Debt

- type (for example, design, testing, or documentation debt),
- intentionality,
- time horizon (short or long term), and
- degree of focus and strategy.

Most definitions, however, come down to some tradeoff among quality, time, and cost. The metaphor implies that, as long as a project properly manages technical debt, it can help that project achieve goals sooner than would have been possible otherwise.

Managing technical debt involves tracking it, making reasoned decisions

Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton*
Carnegie Mellon University Software Engineering Institute
Pittsburgh, PA, USA
{nernst,sbellomo,ozkaya,rn,igorton}@sei.cmu.edu

ABSTRACT

The technical debt metaphor is widely used to encapsulate numerous software quality problems. The metaphor is attractive to practitioners as it communicates to both technical and nontechnical audiences that if quality problems are not addressed, things may get worse. However, it is unclear whether there are practices that move this metaphor beyond a mere communication mechanism. Existing studies of technical debt have largely focused on code metrics and small surveys of developers. In this paper, we report on our survey of 1,831 participants, primarily software engineers and architects working in long-lived, software-intensive projects from three large organizations, and follow-up interviews of seven software engineers. We analyzed our data using both nonparametric statistics and qualitative text analysis. We found that architectural decisions are the most important source of technical debt. Furthermore, while respondents believe the metaphor is itself important for communication, existing tools are not currently helpful in managing the details. We use our results to motivate a technical debt timeline to focus management and tooling approaches.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Management–software development, software selection

Keywords

Technical debt, architecture, survey

1. INTRODUCTION

The technical debt metaphor concisely describes a universal problem that software engineers face when developing software: how to balance near-term value with long-term quality. The appeal is that once technical debt is made visible, engineers (and eventually, their managers) can begin to understand in what ways debt can be harmful or beneficial to a project. Debt accumulates and causes ongoing costs ("interest") to system quality in maintenance and evolution. Debt can be taken on deliberately, then monitored and managed ("principal repaid"), to achieve business value.

The usefulness of this concept prompted the software engineering research community, software consultants, and tool vendors alike to pay more attention to understanding what constitutes technical debt and how to measure, manage, and communicate technical debt. Recent systematic literature reviews [19, 33] report that

- using code quality analysis techniques to understand technical debt has been the dominant focus in research and by tool vendors.
- beyond code quality, other work explores the suitability of the metaphor in other phases of the software life cycle: for example, "requirements debt," "testing debt," "code debt," and "design debt."

Practitioners currently use the term *technical debt* to mean, broadly, a "shortcut for expediency" [23] and, more specifically, bad code or inadequate refactoring [15]. Shull et al. [29], in a review paper on research in technical debt, highlight that technical debt is a multi-faceted problem. Addressing it effectively in practice requires research in software evolution, risk management, qualitative assessment of context, software metrics, program analysis, and software quality. Applying technical debt research starts with identifying a project's sources of "pain." In order to give guidance to a specific project, research in this area must be grounded in the project's context.

This combination of diverse definitions of technical debt in research, alongside the need to ground research on technical debt in practice, raised three research questions:

1. To what extent do practitioners have a commonly shared definition of technical debt?
2. How much of technical debt is architectural in nature?
3. What management practices and tools are used in industry to deal with debt?

To investigate these questions, we conducted a two-part study. First, we administered a survey of software professionals in three large organizations, with 1,831 responses; second, we held semistructured, follow-up interviews with seven respondents, all professional software engineers, to further investigate the emerging themes.

*Now at College of Computer and Information Science, Northeastern University, i.gorton@neu.edu

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786848

… awareness is an important problem in its management.

Developers "self-admit" technical debt…

An Exploratory Study on
Self-Admitted Technical Debt

Aniket Potdar
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
Email: asp6719@rit.edu

Emad Shihab
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
Email: eshihab@cse.concordia.ca

***Abstract*—Throughout a software development life cycle, developers knowingly commit code that is either incomplete, requires rework, produces errors, or is a temporary workaround. Such incomplete or temporary workarounds are commonly referred to as "technical debt". Our experience indicates that self-admitted technical debt is common in software projects and may negatively impact software maintenance, however, to date very little is known about them.**

Therefore, in this paper, we use source-code comments in four large open source software projects - Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML to identify self-admitted technical debt. Using the identified technical debt, we study 1) the amount of self-admitted technical debt found in these projects, 2) why this self-admitted technical debt was introduced into the software projects and 3) how likely is the self-admitted technical debt to be removed after their introduction. We find that the amount of self-admitted technical debt exists in 2.4% - 31% of the files. Furthermore, we find that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt. Lastly, although self-admitted technical debt is meant to be addressed or removed in the future, only between 26.3% - 63.8% of self-admitted technical debt gets removed from projects after introduction.

I. Introduction

Delivering high quality, defect-free software is the goal of all software projects. To ensure the delivery of high quality software, software project often plan their development and maintenance efforts. However, in many cases, developers are rushed into completing tasks for various reasons. A few of these reasons mentioned in prior work include, cost reduction, satisfying customers and market pressure from competition [1]. Intuition and general belief indicate that such rushed development tasks (also known as technical debt) negatively impact software maintenance and overall quality [2].

A plethora of prior work proposed techniques to support software maintenance and ensure high software quality. For example, prior work focused on understanding and predicting software defects (e.g. [3]), analyzing bug fix patterns (e.g. [4]), and attempting to understand and eliminate rework and maintenance (e.g., [5]). The majority of the aforementioned prior work used historical development data and source-code metrics to perform their studies. More recently, researchers leveraged natural language to help identify potentially problematic areas of the software. For example, work by Tan *et al.* [6] developed natural language processing tools to find comment-bug inconsistencies. Other work identified the coevolutionary relationship between source code and its associated comments (e.g., [7], [8]) and used task annotations to manage productivity [9].

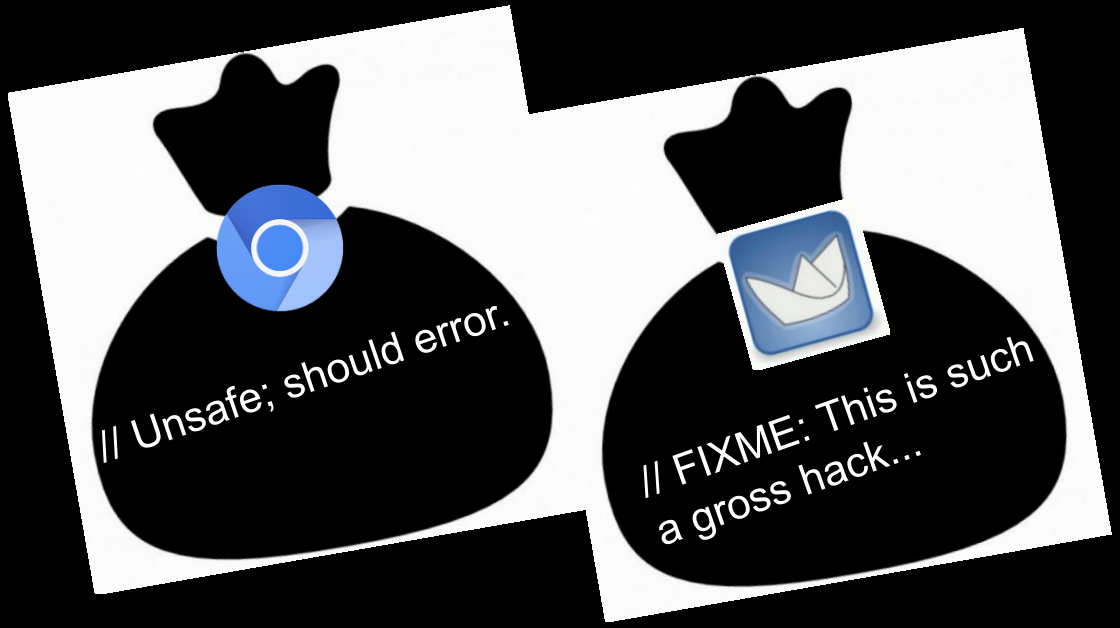
The majority of the prior work focused on quality issues that are due to *unintentional* errors by developers (i.e., errors introduced by the developers are assumed to mistakes). However, to the best of our knowledge, very few prior studies examined the impact of errors that might be introduced due to *intentional* (i.e., self admitted) quick or temporary fixes (i.e., technical debt). Studying this self-admitted technical debt is important since they appear frequently in some projects (as we show later in this study) and prior work indicated that they negatively impact quality [3].

Therefore, in this paper we perform an exploratory study to better understand self-admitted technical debt. Inspired by prior work (e.g., [6], [8], [10], [11]), we use source-code comments to detect self-admitted technical debt. We perform our study on four large open source projects - namely Eclipse, Chromium OS, ArgoUML, and Apache httpd. We focus on quantifying the amount of self-admitted technical debt (RQ1), on determining why self-admitted technical debt is introduced (RQ2) and how much of self-admitted technical debt is actually removed after their introduction (RQ3).

We make the following contributions:

- **Identify comment patterns that indicate self-admitted technical debt.** We manually read through 101,762 code comments to determine patterns that indicated self-admitted technical debt. In the end, we identified 62 different comment patterns that indicate self-admitted technical debt.
- **Measure how much self-admitted technical debt exists, why self-admitted technical debt is introduced and how much self-admitted technical debt is removed after their introduction.** We find that 2.4% - 31.0% of the files contain self-admitted technical debt, that more experienced developers introduce more self-admitted technical debt and that self-admitted technical debt is introduced throughout their development activity (i.e., they do not only introduce self-admitted technical

"… at the file level, between 2.4 - 31.0% of the files contained one or more instances of self-admitted technical debt."



Detecting and Quantifying Different Types of Self-Admitted Technical Debt

Everton da S. Maldonado and Emad Shihab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{e_silvam,eshihab}@encs.concordia.ca

Abstract—Technical Debt is a term that has been used to express non-optimal s projects. These non allow the project to of increased mainten impact of technical detection of technica that one possible sou code comments, also However, what types source code comment Therefore, in this determine the differe four simple filtering h likely to contain tech than 33K comments, debt can be classified debt, documentation most common type of making up between Lastly, we make the cl publicly available for research and the evol

The software deve There are short dea be made, high qual environment. Often than time to accomp under increasing pr achieving many conf decisions are made t project at the cost o future. This phenom With the organiz through the managi work has focused on studying the impact of technical debt in many efforts to dete challenge [3]. One debt is self-admitte reported in source debt refers to the si current implementati alerting the inadequa

Recently, Potdar and Shihab [3] developed an approach

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among those, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a *differentiated replication* of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 51 instances per system, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•**Software and its engineering** → **Software evolution; Maintaining software;**

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

MSR'16, May 14-15, 2016, Austin, TX, USA

 ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901742

1. INTRODUCTION

Ward Cunningham coined the technical debt metaphor back in 1993 [14] to explain the unavoidable interests (*i.e.*, maintenance and evolution costs) developers pay while working on *not-quite-right code*, possibly written in a rush to meet a deadline or to deliver the software to the market in the shortest time possible [7, 14, 22, 26, 33].

In the last years researchers have studied the technical debt phenomenon from different perspectives. Several authors developed techniques and tools aimed at detecting specific types of technical debt, like code smells (see *e.g.*, [28, 29]) and coding style violations (see *e.g.*, [1]). Also, researchers have studied the impact of different types of technical debt on maintainability attributes of software systems [20, 40] and when and why technical debt instances are introduced in software systems [36].

Recently, Potdar and Shihab [30] pioneered the study of *self-admitted technical debt* (SATD), referring to technical debt instances intentionally introduced by developers (*e.g.*, temporary patches to fix bugs) and explicitly documented in code comments. They showed how it is possible to detect instances of technical debt by simply mining code comments looking for patterns likely indicating (*i.e.*, self-admitting) the presence of technical debt (*e.g., fixme, todo, to be fixed, etc.*). They performed an exploratory study on five software systems aimed at understanding the diffusion of SATD, the factors promoting its introduction, and the amount of technical debt removed by developers after its introduction. Maldonado and Shihab [15] also showed that SATD can be exploited to identify several different types of technical debt (*e.g.*, design and code debt), while Wehaibi *et al.* [37] highlighted that the presence of SATD makes the code more difficult to change in the future. Despite these studies, there is still a noticeable lack of empirical evidence related to the magnitude of the phenomenon, its evolution over time, and its relationship with software quality. This represents an obstacle for an effective and efficient management of technical debt.

In this paper we contribute to enlarge the knowledge about the technical debt phenomenon by presenting a large-scale empirical study on SATD, as a *differentiated replication* [38] of the study by Potdar and Shihab [30][1]. In particular, we mine the complete change history of 159 Java open source systems to detect SATD instances across over 600K commits, for a total of over 2 Billion comments mined. Using these data we analyse the diffusion of SATD (*i.e.*, the number of SATD

[1]Similarities and differences between the two studies are carefully described in Section 5 while discussing the related literature.

Code Debt

Documentation Debt

Defect Debt

Requirement Debt

Test Debt

Design Debt

Design Debt

Detecting and Quantifying Different Types of Self-Admitted Technical Debt

Everton da S. Maldonado and Emad Shihab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{e_silvam,eshihab}@encs.concordia.ca

Abstract—Technical Debt is a term that has been used to express non-optimal s projects. These non allow the project to of increased mainten impact of technical detection of technica that one possible sou code comments, also However, what types source code comment Therefore, in this determine the differe four simple filtering h likely to contain tech

Recently, Potdar and Shihab [3] developed an approach

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2654244, IEEE Transactions on Software Engineering

Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

Everton da S. Maldonado, Emad Shihab, *Member, IEEE,* and Nikolaos Tsantalis, *Member, IEEE,*

Abstract—The metaphor of technical debt was introduced to express the trade off between productivity and quality, i.e., when developers take shortcuts or perform quick hacks. More recently, our work has shown that it is possible to detect technical debt using source code comments (i.e., self-admitted technical debt), and that the most common types of self-admitted technical debt are design and requirement debt. However, all approaches thus far heavily depend on the manual classification of source code comments. In this paper, we present an approach to automatically identify design and requirement self-admitted technical debt using Natural Language Processing (NLP). We study 10 open source projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL and find that 1) we are able to accurately identify self-admitted technical debt, significantly outperforming the current state-of-the-art based on fixed keywords and phrases; 2) words related to sloppy code or mediocre source code quality are the best indicators of design debt, whereas words related to the need to complete a partially implemented requirement in the future are the best indicators of requirement debt; and 3) we can achieve 90% of the best classification performance, using as little as 23% of the comments for both design and requirement self-admitted technical debt, and 80% of the best performance, using as little as 9% and 5% of the comments for design and requirement self-admitted technical debt, respectively. The last finding shows that the proposed approach can achieve a good accuracy even with a relatively small training dataset.

Index Terms—Technical debt, Source code comments, Natural language processing, Empirical study.

"… design debt comments represent the majority of self-admitted technical debt comments…"

Design Debt

"… can be discovered by analyzing the source code…"

Detecting and Quantifying Different Types of Self-Admitted Technical Debt

Everton da S. Maldonado and Emad Shihab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{e_alvare,eshihab}@cse.concordia.ca

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among those, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a *differentiated replication* of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 51 instances per system, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•Software and its engineering → Software evolution; Maintaining software;

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

1. INTRODUCTION

Ward Cunningham coined the technical debt metaphor back in 1993 [14] to explain the unavoidable interests (*i.e.*, maintenance and evolution costs) developers pay while working on *not-quite-right code*, possibly written in a rush to meet a deadline or to deliver the software to the market in the shortest time possible [7, 14, 22, 26, 33].

In the last years researchers have studied the technical debt phenomenon from different perspectives. Several authors developed techniques and tools aimed at detecting specific types of technical debt, like code smells (see *e.g.*, [28, 29]) and coding style violations (see *e.g.*, [1]). Also, researchers have studied the impact of different types of technical debt on maintainability attributes of software systems [20, 40] and when and why technical debt instances are introduced in software systems [36].

Recently, Potdar and Shihab [30] pioneered the study of *self-admitted technical debt* (SATD), referring to technical debt instances intentionally introduced by developers (*e.g.*, temporary patches to fix bugs) and explicitly documented in code comments. They showed how it is possible to detect instances of technical debt by simply mining code comments looking for patterns likely indicating (*i.e.*, self-admitting) the presence of technical debt (*e.g.*, *fixme*, *todo*, *to be fixed*, *etc.*). They performed an exploratory study on five software systems aimed at understanding the diffusion of SATD, the factors promoting its introduction, and the amount of technical debt removed by developers after its introduction. Maldonado and Shihab [15] also showed that SATD can be exploited to identify several different types of technical debt (*e.g.*, design and code debt), while Wehaibi *et al.* [37] highlighted that the presence of SATD makes the code more difficult to change in the future. Despite these studies, there is still a noticeable lack of empirical evidence related to the magnitude of the phenomenon, its evolution over time, and its relationship with software quality. This represents an obstacle for an effective and efficient management of technical debt.

In this paper we contribute to enlarge the knowledge about the technical debt phenomenon by presenting a large-scale empirical study on SATD, as a *differentiated replication* [38] of the study by Potdar and Shihab [30][1]. In particular, we mine the complete change history of 159 Java open source systems to detect SATD instances across over 600K commits, for a total of over 2 Billion comments mined. Using these data we analyse the diffusion of SATD (*i.e.*, the number of SATD

[1]Similarities and differences between the two studies are carefully described in Section 5 while discussing the related literature.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR'16, May 14-15, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901742

315

// FIXME: to be moved to ApiResponseHelper

// TODO: There is overlap between ValidTypeMap and PrimitiveTypeName in parquet-mr, it might be a good idea to unify these two classes

// FIXME extract some method

Is it possible to recommend developers with "design technical debt to be admitted"?

TEDIOuS

(TEchnical Debt IdentificatiOn System)

```
#define zb69998dcc6
int z1c0ab7cf0c, z8c8f7c7867, za862d19cbc= (0x6d8+5427-0x1bbb); int main(
int za82b547bcb, char* z6965940303[ ] ) { int z2d29194d43; char z34a935337a[
(0x138f+2785-0x1e20)] ; while( ( z2d29194d43= zefd3fb4f7d( za82b547bcb,
z6965940303, "\x6e\x3a\x6f\x3a\x76\x56\x3a", ((char)(0x2053+885-0x2360)), ((cha
(0x2368+457-0x24db))) ) != - (0xc4+9243-0x24de)) { switch( z2d29194d43) {
case((char)(0x8f4+4043-0x1851)): z1c0ab7cf0c= z048b31e7a8( atoi( zbb0e42ea6d)
(0xfec+5036-0x2299)) ; break; case((char)(0x106+7700-0x1eab)): if( freopen(
zbb0e42ea6d, "\x77", stdout) == (0xf93+3040-0x1b73)) z6aea0a920d( zbb0e42ea6
; break; case((char)(0x8b7+5968-0x1f91)): z8c8f7c7867= 1;
  break;
case ((char)(0x88f+5161-0x1c62)):
#ifndef NO_IDENT
  if ( ! sscanf( id, "%*s %*s %s", version) )
```

TEDIOuS

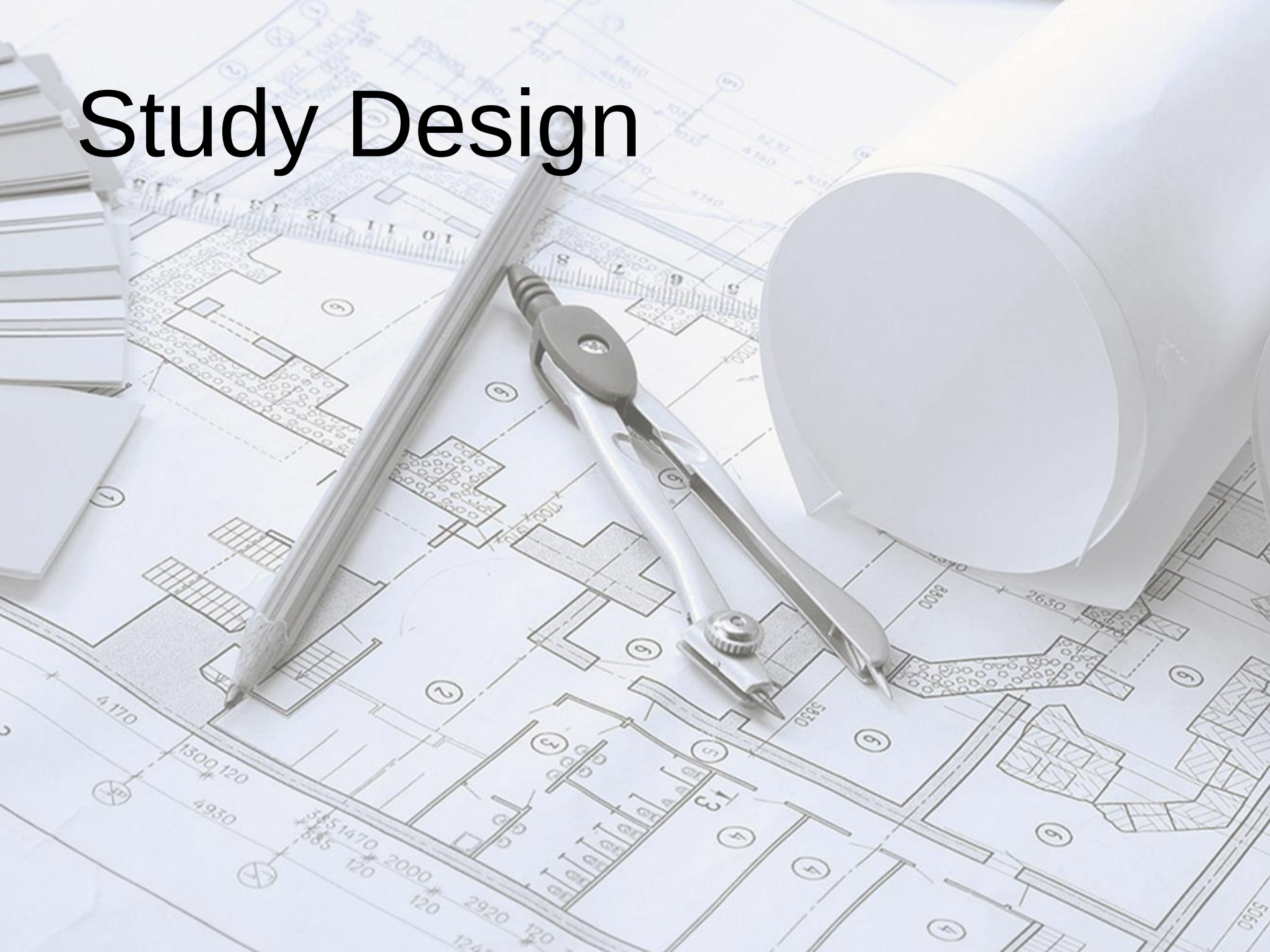
(TEchnical Debt IdentificatiOn System)

RQ_1: Performance within-project

RQ_2: Performance across-project

RQ_3: TEDIOuS *vs* method-level smell detector

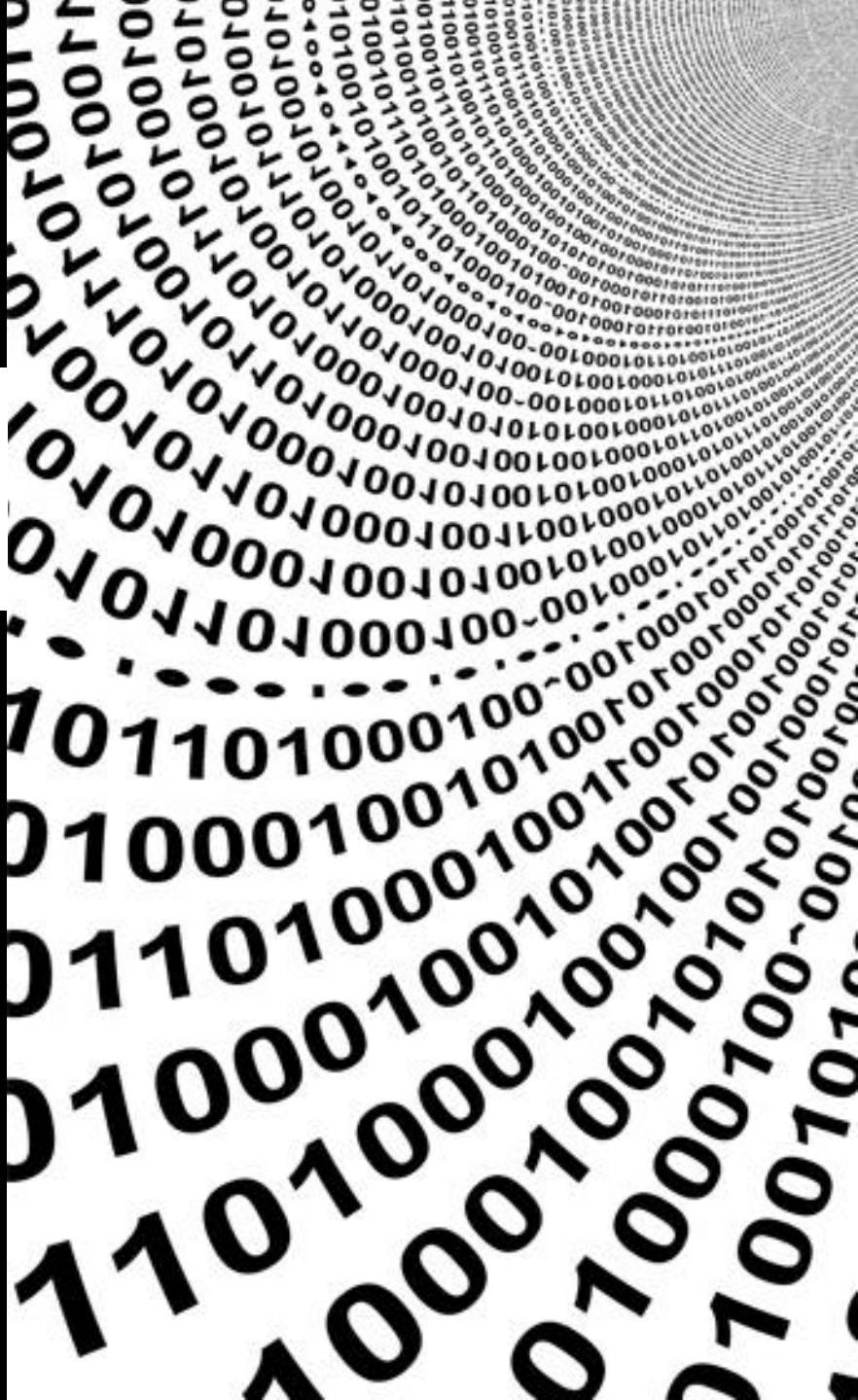
Study Design



Study Context

Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

Everton da S. Maldonado, Emad Shihab, *Member, IEEE,* and Nikolaos Tsantalis, *Member, IEEE,*



Study Context

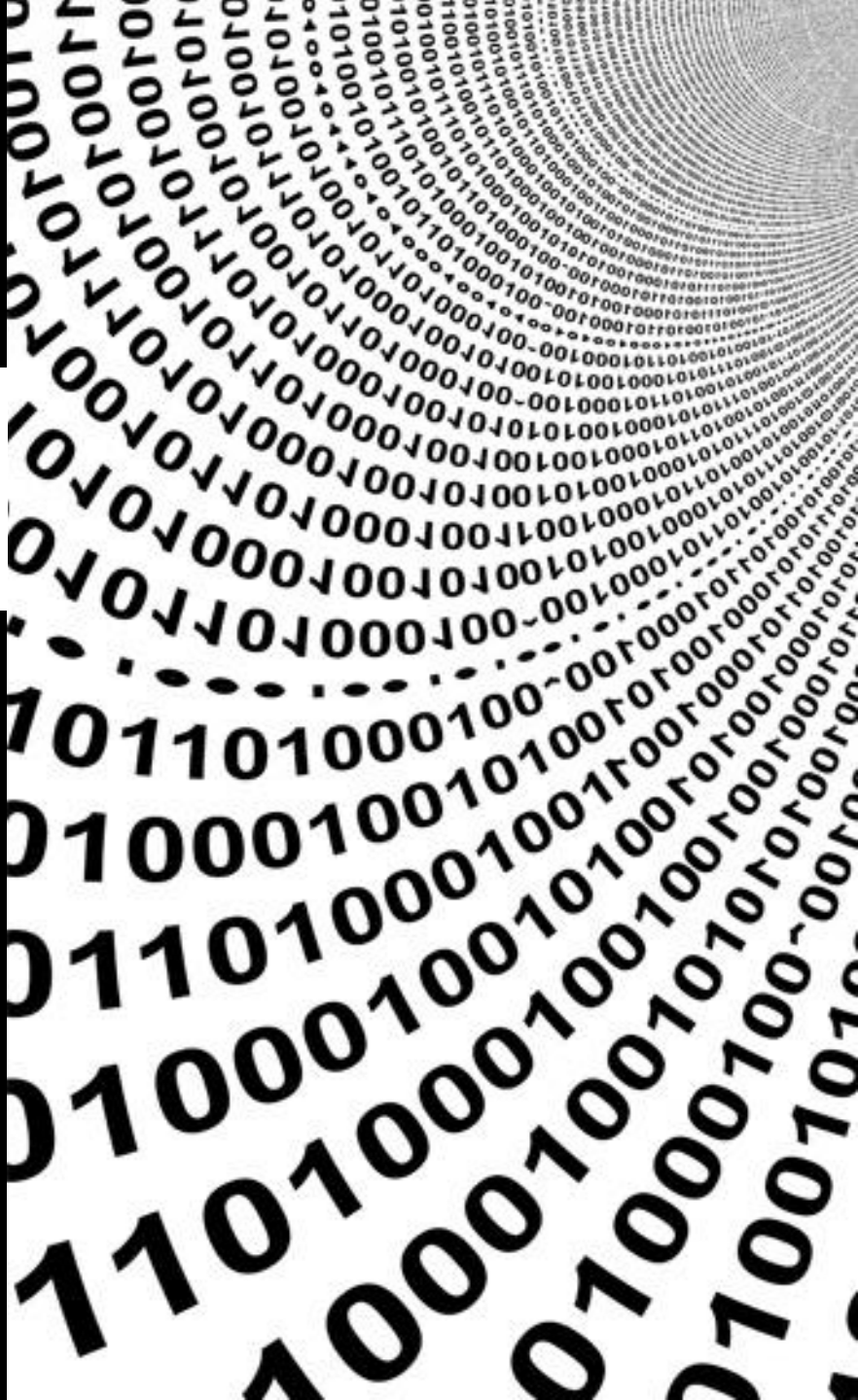
Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

Everton da S. Maldonado, Emad Shihab, *Member, IEEE,* and Nikolaos Tsantalis, *Member, IEEE,*

9 OSS with validated SATD

% of design SATD < 0.5

Classification problem is <u>unbalanced</u>

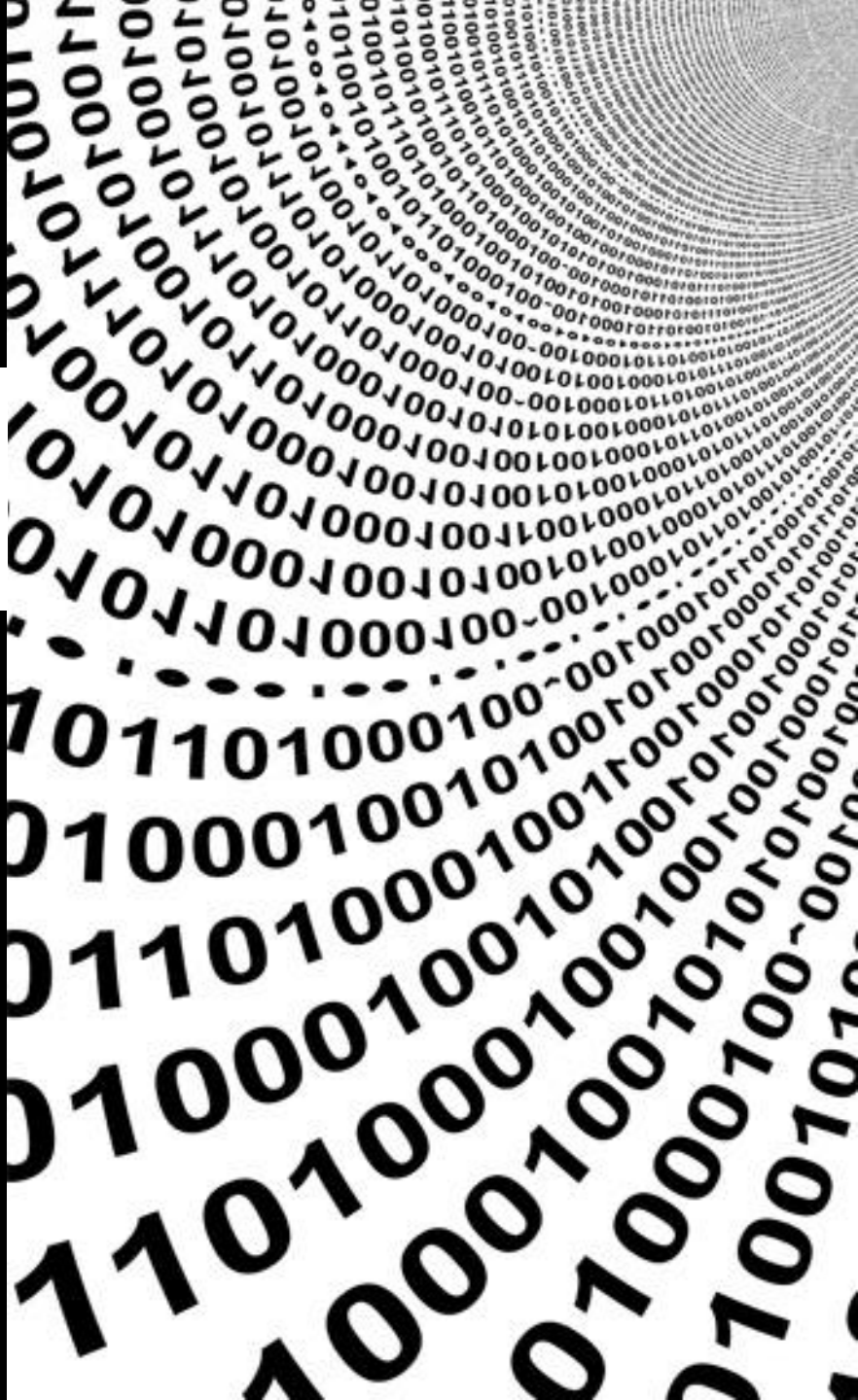


Study Context

Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

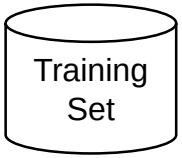
Everton da S. Maldonado, Emad Shihab, *Member, IEEE,* and Nikolaos Tsantalis, *Member, IEEE,*

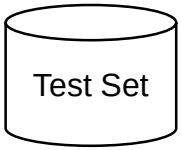
Project	# of Methods	Design SATD	% of Methods with SATD
Ant	11052	57	0.5%
ArgoUML	14346	425	2%
Columba	7035	418	5%
Hibernate	17405	377	2%
jEdit	4785	77	2%
jFreeChart	10343	1881	18%
jMeter	8680	424	5%
jRuby	14163	275	2%
Squirrel	16648	173	1%



TEDIOuS

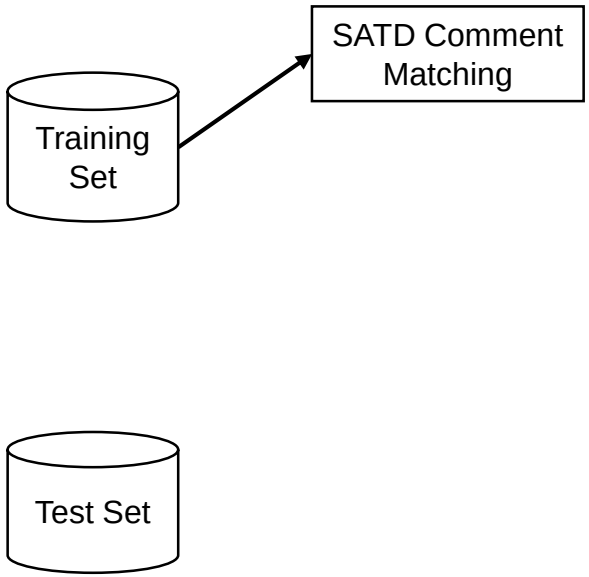
(TEchnical Debt IdentificatiOn System)





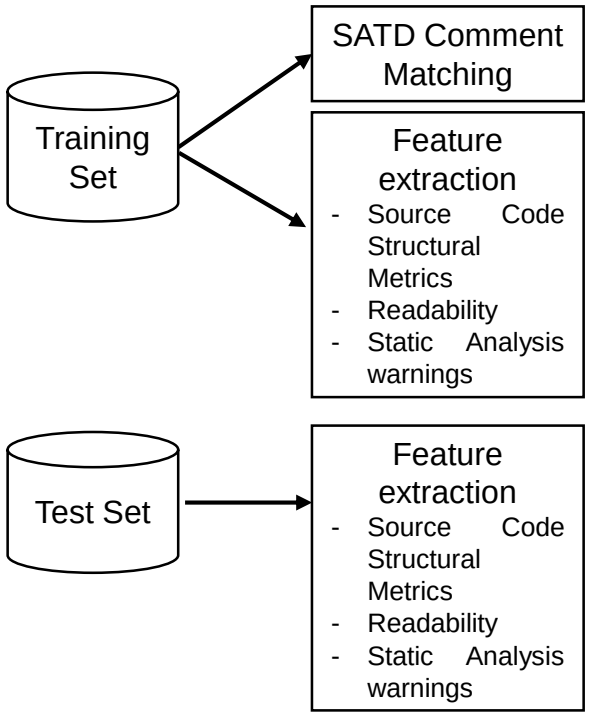
TEDIOuS

(TEchnical Debt IdentificatiOn System)



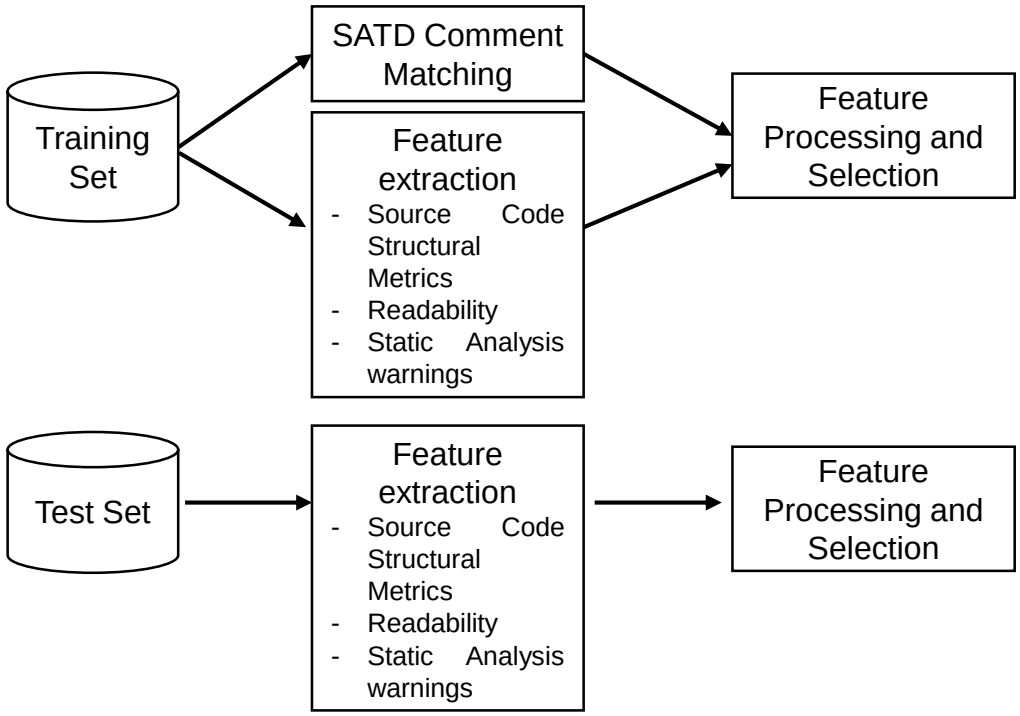
TEDIOuS

(TEchnical Debt IdentificatiOn System)



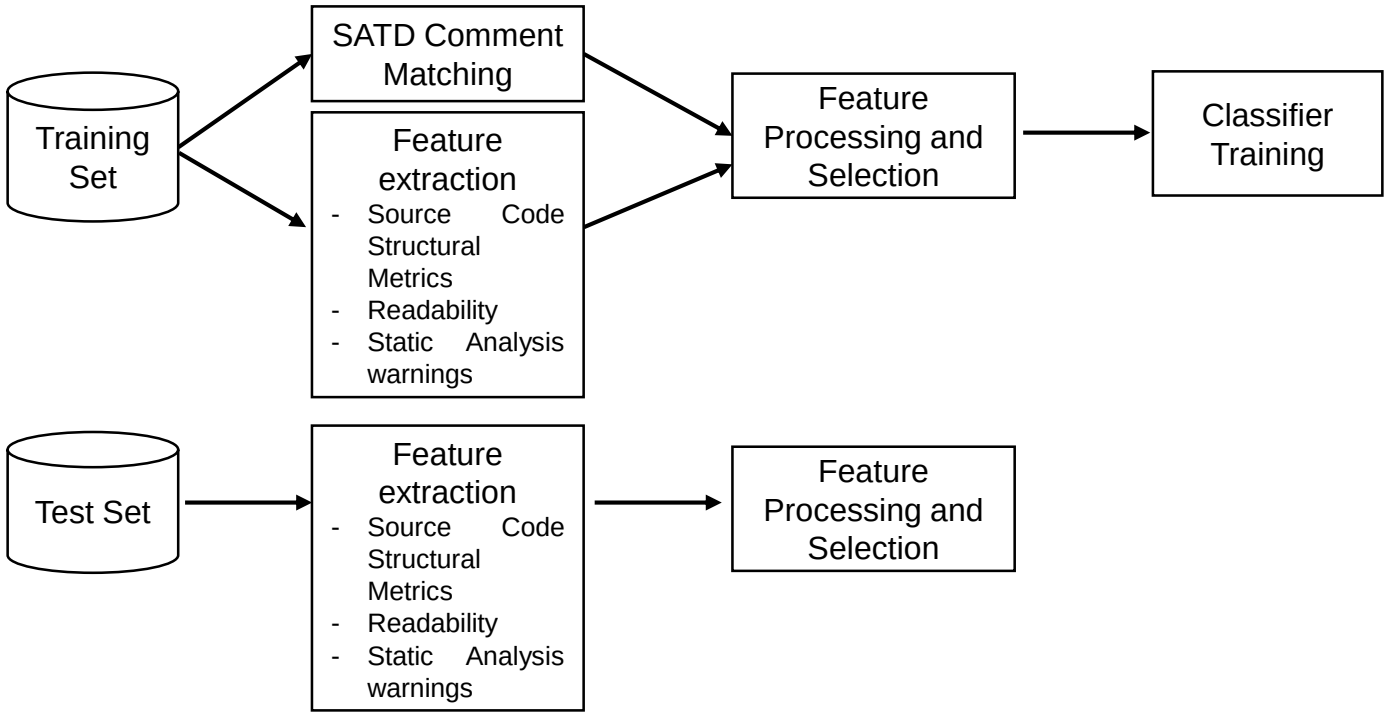
TEDIOuS

(TEchnical Debt IdentificatiOn System)



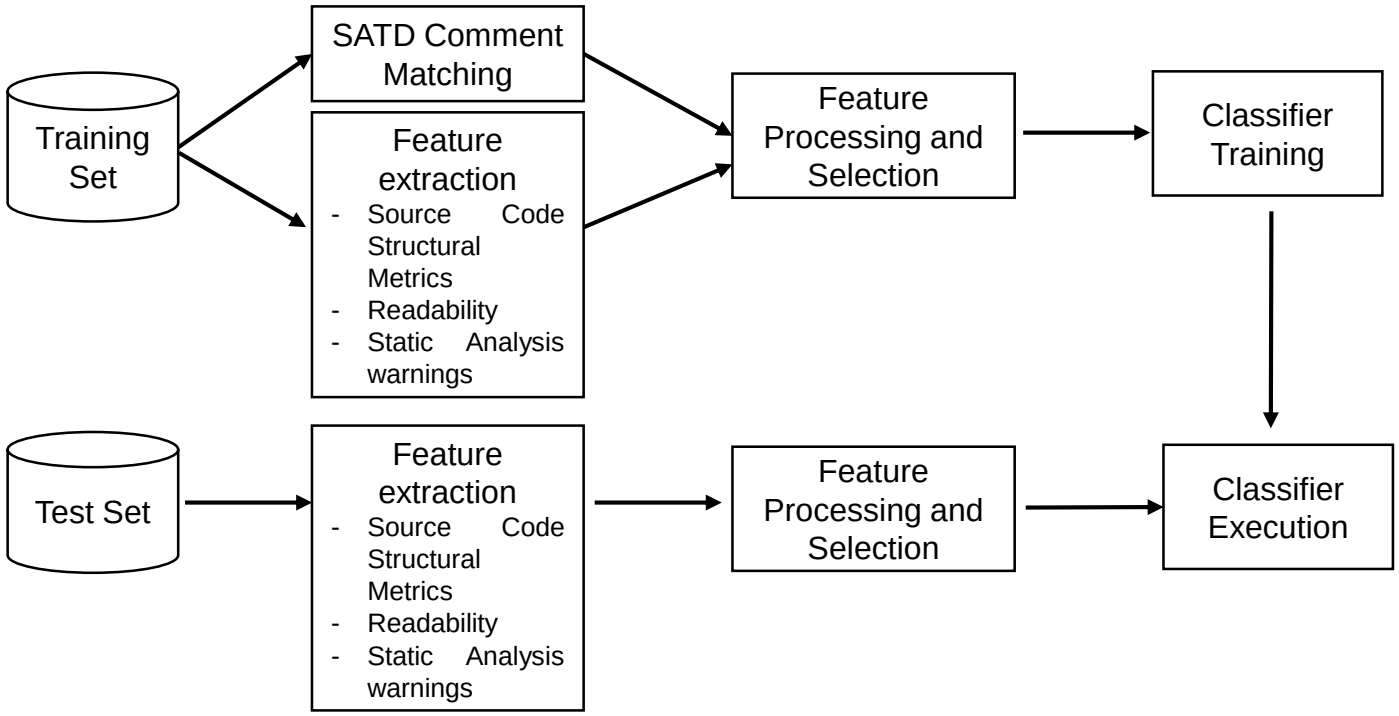
TEDIOuS

(TEchnical Debt IdentificatiOn System)



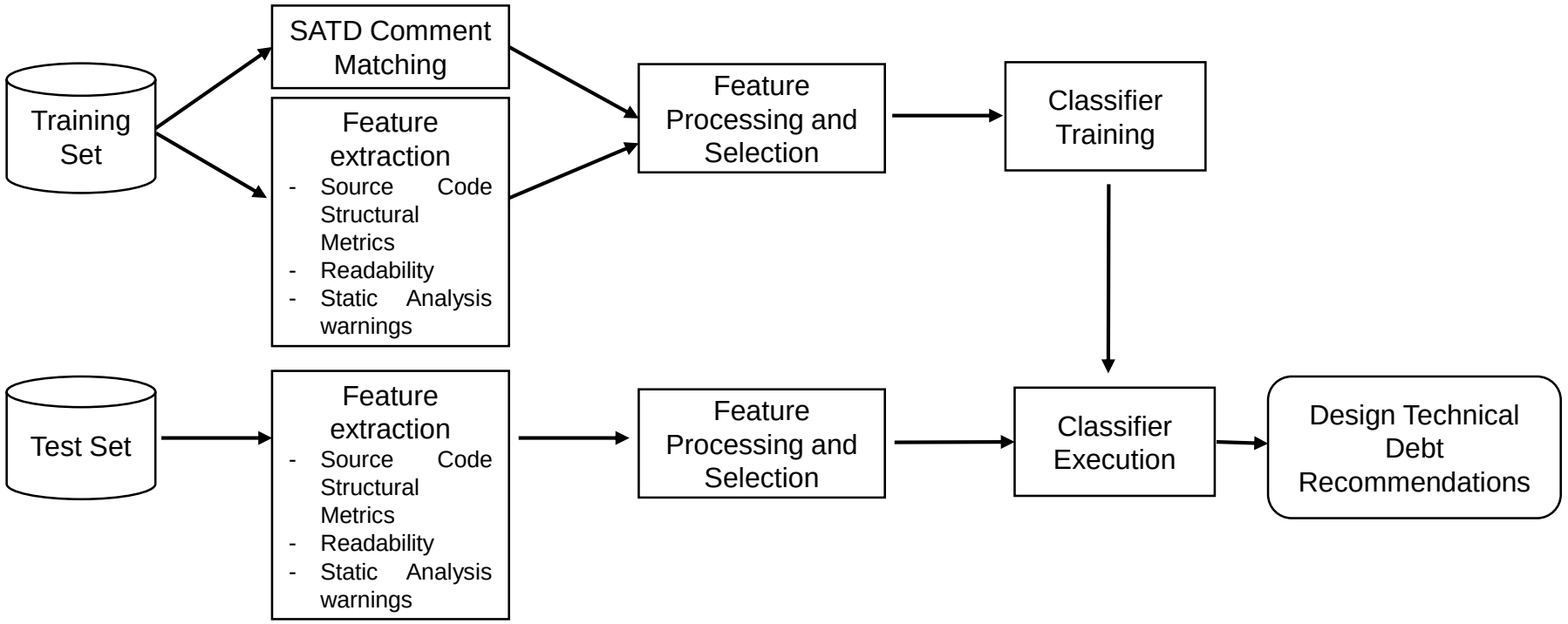
TEDIOuS

(TEchnical Debt IdentificatiOn System)



TEDIOuS

(TEchnical Debt IdentificatiOn System)



Study Results

Analysis Method

RQ_1 & RQ_2: Performance within/across project

$$Precision(Pr) = \frac{TP}{TP + FP}$$

$$Recall(Rc) = \frac{TP}{TP + FN}$$

$$F_1 = 2\frac{Pr \cdot Rc}{Pr + Rc}$$

Analysis Method

RQ_1 & RQ_2: Performance within/across project

$$Precision(Pr) = \frac{TP}{TP + FP}$$

$$Accuracy(Acc) = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Recall(Rc) = \frac{TP}{TP + FN}$$

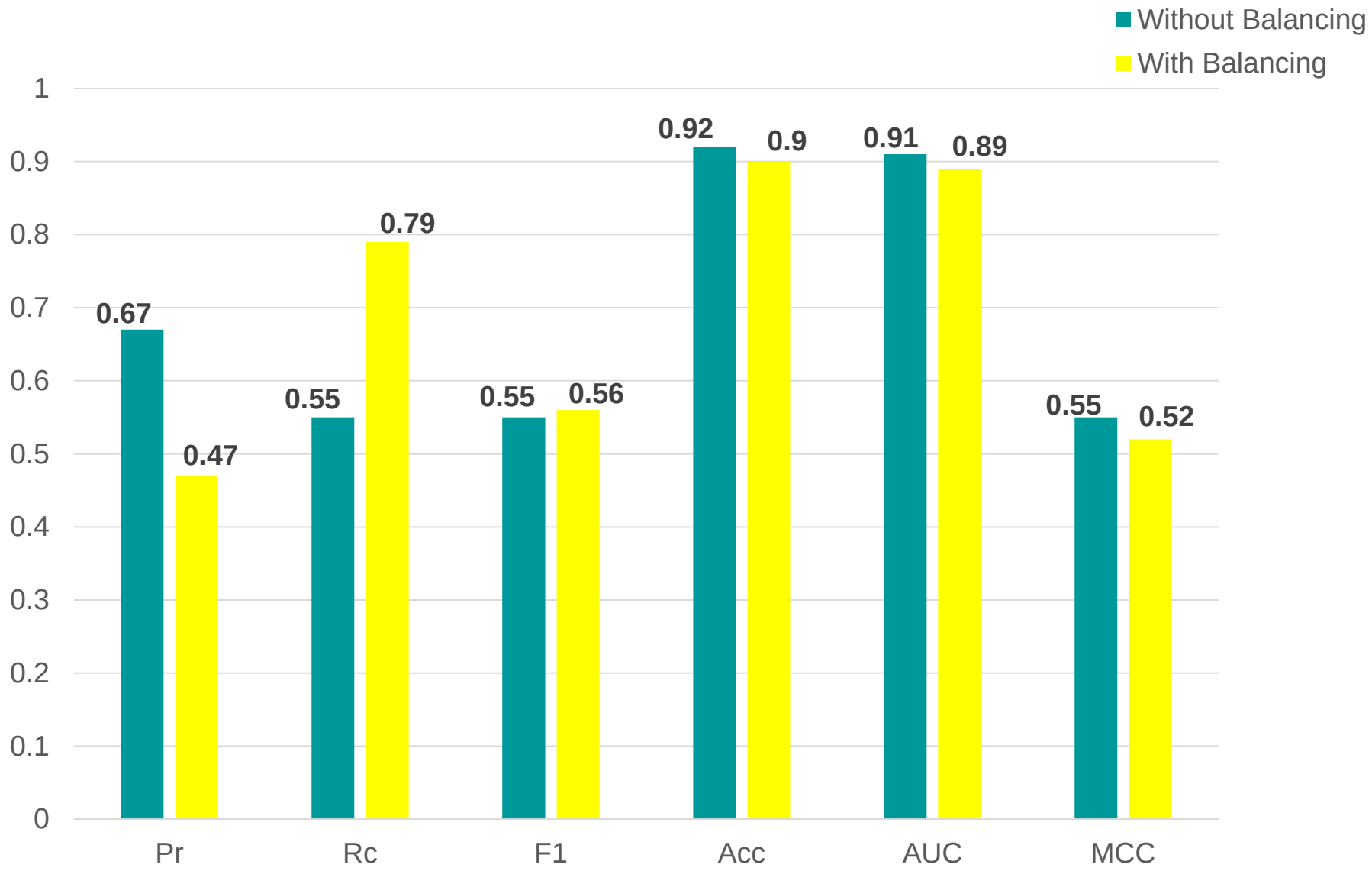
$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(FN + TN)(FP + TN)(TP + FN)}}$$

$$F_1 = 2\frac{Pr \cdot Rc}{Pr + Rc}$$

$$AUC \neq 0.5$$

RQ_1: Performance within-project

RQ_2: Performance across-project



Analysis Method

RQ_1 & RQ_2: Performance within/across project

Measure the importance of variables on an ensemble of randomized trees.
MDI (Mean Decrease Impurity)

RQ_1 & RQ_2: Performance within/across project

```
// FIXME: If true array is common enough we should pre-allocate and stick somewhere
private IRubyObject[] trueIfNoArgument (ThreadContext context, IRubyObject[] args) {
        return args.length == 0 ? new IRubyObject[] {context.getRuntime().getTrue() } : args; }
```

Metric	Value
Readability	0.94
LOC	2
McCabe	1

Line Length (CheckStyle)

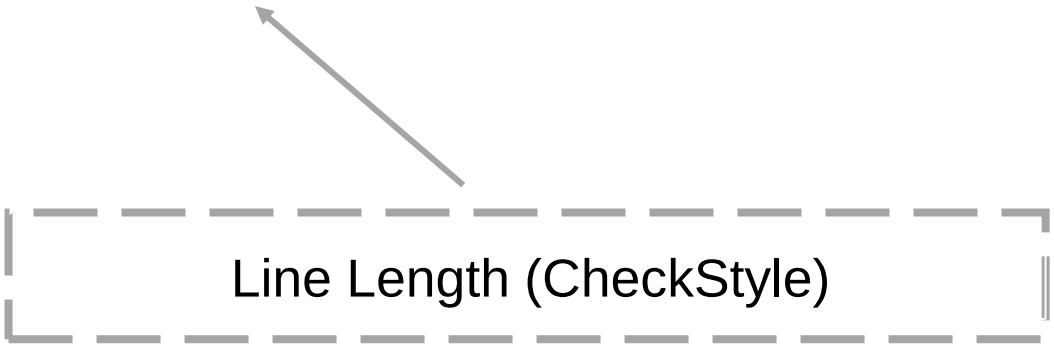
Method Argument Could be Final (PMD)

RQ_1 & RQ_2: Performance within/across project

```
// FIXME: this should go somewhere more generic -- maybe IdUtil

public static final boolean isRubyVariable (String name) {
        char c;
        return name.length() > 0 && ((c = name.charAt(0)) == '@' || (c <= 'Z' && c >= 'A')); }
```

Metric	Value
Readability	0.74
LOC	3
McCabe	1



RQ_1 & RQ_2: Performance within/across project

```
/**
* Arranges the blocks in the container with the overall height and width specified as fixed
* constraints.
*
* @param container the container.
* @param constraint the constraint.
* @param g2 the graphics device.
*
* @return The size following the arrangement.
*/
protected Size2D arrangeFF(BlockContainer container, Graphics2D g2, RectangleConstraint constraint)
{
        // TODO: implement this properly
        return arrangeFN(container, g2, constraint);
}
```

Metric	Value
Readability	0.98
LOC	5
McCabe	1

Short Variable (PMD)

RQ_1 & RQ_2: Performance within/across project

```
public void exportMethod(String name, Visibility visibility) {
        if (this == getRuntime().getObject()) {
                getRuntime().secure(4);
        }
         DynamicMethod method = searchMethod(name);
        if (method.isUndefined()) {
                throw getRuntime().newNameError("undefined method '" + name + "' for " + (isModule() ?
                "module" : "class") + " '" + getName() + "'", name);
         }
        if (method.getVisibility() != visibility) {
                if (this == method.getImplementationClass()) {
                        method.setVisibility(visibility);
                } else {
                 // FIXME: Why was this using a FullFunctionCallbackMethod before that did callSuper?
                        addMethod(name, new WrapperMethod(this, method, visibility));
                }
                invalidateCacheDescendants();
        }
}
```

Metric	Value
Readability	0.08
LOC	21
McCabe	5
# of Calls	16

RQ_1 & RQ_2: Performance within/across project

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among those, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a *differentiated replication* of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 51 instances per system, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•**Software and its engineering** → **Software evolution; Maintaining software;**

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901742

1. INTRODUCTION

Ward Cunningham coined the technical debt metaphor back in 1993 [14] to explain the unavoidable interests (*i.e.*, maintenance and evolution costs) developers pay while working on *not-quite-right code*, possibly written in a rush to meet a deadline or to deliver the software to the market in the shortest time possible [7, 14, 22, 26, 33].

In the last years researchers have studied the technical debt phenomenon from different perspectives. Several authors developed techniques and tools aimed at detecting specific types of technical debt, like code smells (see *e.g.*, [28, 29]) and coding style violations (see *e.g.*, [1]). Also, researchers have studied the impact of different types of technical debt on maintainability attributes of software systems [20, 40] and when and why technical debt instances are introduced in software systems [30].

Recently, Potdar and Shihab [30] pioneered the study of *self-admitted technical debt* (SATD), referring to technical debt instances intentionally introduced by developers (*e.g.*, temporary patches to fix bugs) and explicitly documented in code comments. They showed how it is possible to detect instances of technical debt by simply mining code comments looking for patterns likely indicating (*i.e.*, self-admitting) the presence of technical debt (*e.g., fixme, todo, to be fixed, etc.*). They performed an exploratory study on five software systems aimed at understanding the diffusion of SATD, the factors promoting its introduction, and the amount of technical debt removed by developers after its introduction. Maldonado and Shihab [15] also showed that SATD can be exploited to identify several different types of technical debt (*e.g.*, design and code debt), while Wehaibi *et al.* [37] highlighted that the presence of SATD makes the code more difficult to change in the future. Despite these studies, there is still a noticeable lack of empirical evidence related to the magnitude of the phenomenon, its evolution over time, and its relationship with software quality. This represents an obstacle for an effective and efficient management of technical debt.

In this paper we contribute to enlarge the knowledge about the technical debt phenomenon by presenting a large-scale empirical study on SATD, as a *differentiated replication* [38] of the study by Potdar and Shihab [30][1]. In particular, we mine the complete change history of 159 Java open source systems to detect SATD instances across over 600K commits, for a total of over 2 Billion comments mined. Using these data we analyse the diffusion of SATD (*i.e.*, the number of SATD

[1]Similarities and differences between the two studies are carefully described in Section 5 while discussing the related literature.

"We found no correlation between the code file internal quality and the number of SATD instances they contain. "

RQ_1 & RQ_2: Performance within/across project

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among those, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a *differentiated replication* of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 51 instances per system, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•**Software and its engineering** → **Software evolution; Maintaining software;**

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

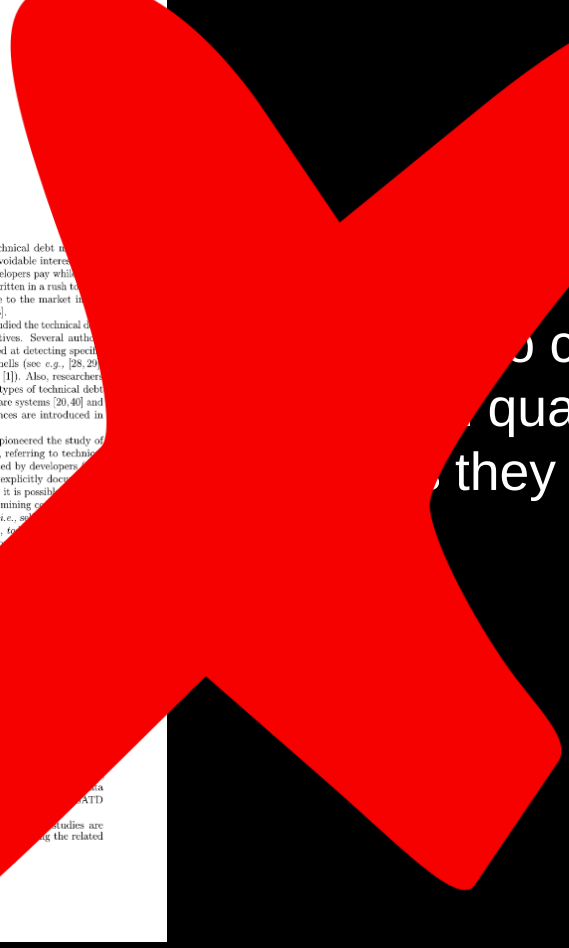
MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901742

1. INTRODUCTION

Ward Cunningham coined the technical debt
back in 1993 [14] to explain the unavoidable
maintenance and evolution costs) developers pay
ing on *not-quite-right code*, possibly written in a rush
a deadline or to deliver the software to the market
shortest time possible [7, 14, 22, 26, 33].

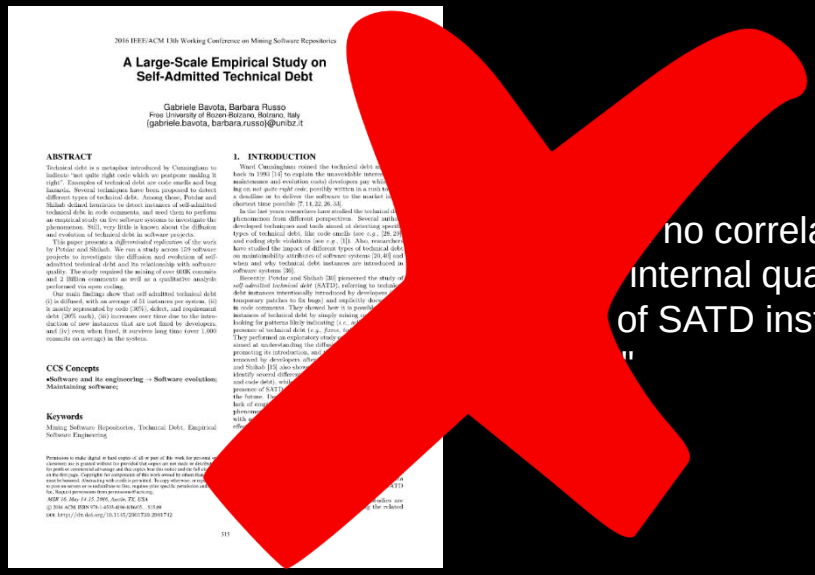
In the last years researchers have studied the technical
phenomenon from different perspectives. Several
developed techniques and tools aimed at detecting
types of technical debt, like code smells (see *e.g.*, [28, 29]
and coding style violations (see *e.g.*, [1]). Also, researchers
have studied the impact of different types of technical debt
on maintainability attributes of software systems [20, 40] and
when and why technical debt instances are introduced in
software systems [30].

Recently, Potdar and Shihab [30] pioneered the study of
self-admitted technical debt (SATD), referring to
debt instances intentionally introduced by developers
temporary patches to fix bugs) and explicitly
in code comments. They showed how it is possible
instances of technical debt by simply mining
looking for patterns likely indicating (*i.e.*,
presence of technical debt (*e.g.*, *fixme*,
They performed an exploratory study
aimed at understanding the diffu
promoting its introduction, and
removed by developers after
and Shihab [15] also show
identify several differen
and code debt), whil
presence of SATD
the future. D
lack of emp
phenom
with s
effo



o correlation between the code
quality and the number of SATD
they contain. "

RQ_1 & RQ_2: Performance within/across project



2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Empirical Study on Self-Admitted Technical Debt

Gabriele Bavota, Barbara Russo
Free University of Bozen-Bolzano, Bolzano, Italy
{gabriele.bavota, barbara.russo}@unibz.it

ABSTRACT

Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". Examples of technical debt are code smells and bug hazards. Several techniques have been proposed to detect different types of technical debt. Among these, Potdar and Shihab defined heuristics to detect instances of self-admitted technical debt, in code comments, and used them to perform an empirical study on five software systems to investigate the phenomenon. Still, very little is known about the diffusion and evolution of technical debt in software projects.

This paper presents a differentiated replication of the work by Potdar and Shihab. We run a study across 159 software projects to investigate the diffusion and evolution of self-admitted technical debt and its relationship with software quality. The study required the mining of over 600K commits and 2 Billion comments as well as a qualitative analysis performed via open coding.

Our main findings show that self-admitted technical debt (i) is diffused, with an average of 31 instances per systems, (ii) is mostly represented by code (30%), defect, and requirement debt (20% each), (iii) increases over time due to the introduction of new instances that are not fixed by developers, and (iv) even when fixed, it survives long time (over 1,000 commits on average) in the system.

CCS Concepts

•**Software and its engineering** → **Software evolution; Maintaining software;**

Keywords

Mining Software Repositories, Technical Debt, Empirical Software Engineering

MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901742

no correlation between the
internal quality and the
of SATD instances they
"

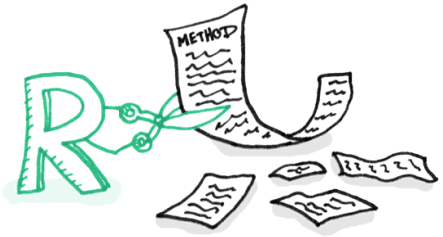
"Working at method level there is a relation between the internal quality and the number of Design SATD instances in it."

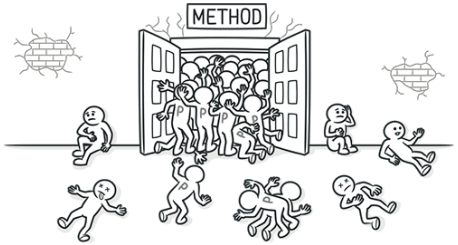
Analysis Method

RQ_3: TEDIOuS vs method-level smell detector

DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur





$LOC > \boldsymbol{th_1}$ $ParNbr > \boldsymbol{th_2}$

RQ_3: TEDIOuS vs method-level smell detector

F1 never exceeds 22%

MCC never exceeds 0.17 ➡ Low correlation

RQ_3: TEDIOuS vs method-level smell detector

F1 never exceeds 22%

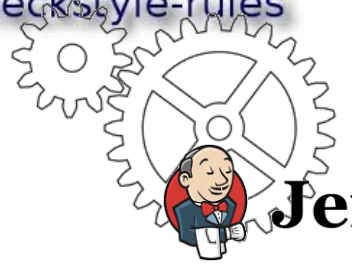
MCC never exceeds 0.17 ➡ Low correlation

	F1	MCC
Long Method	21%	0.17
TEDIOuS	47.15%	0.47

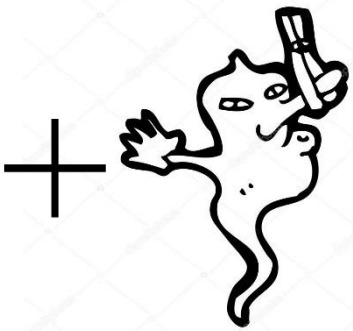
What else?

TEDIOuS usage scenarios

checkstyle-rules



Jenkins

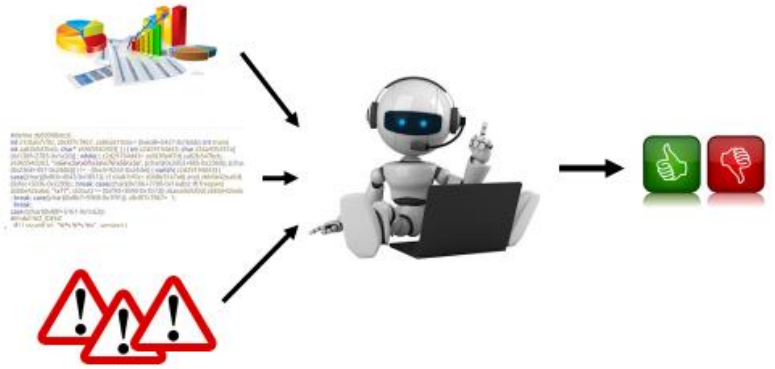


Conclusions

Conclusions

TEDIOuS

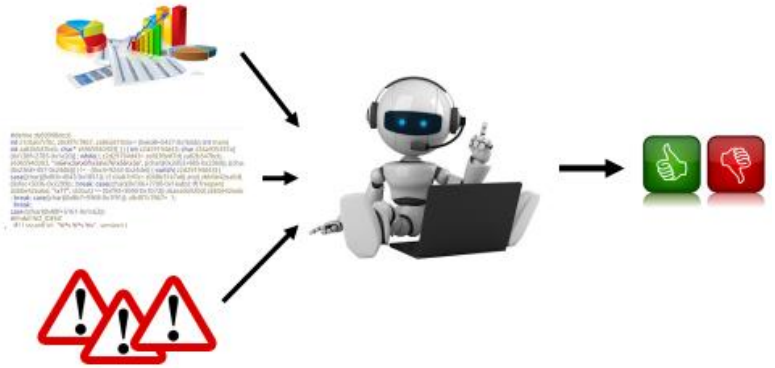
(TEchnical Debt IdentificatiOn System)



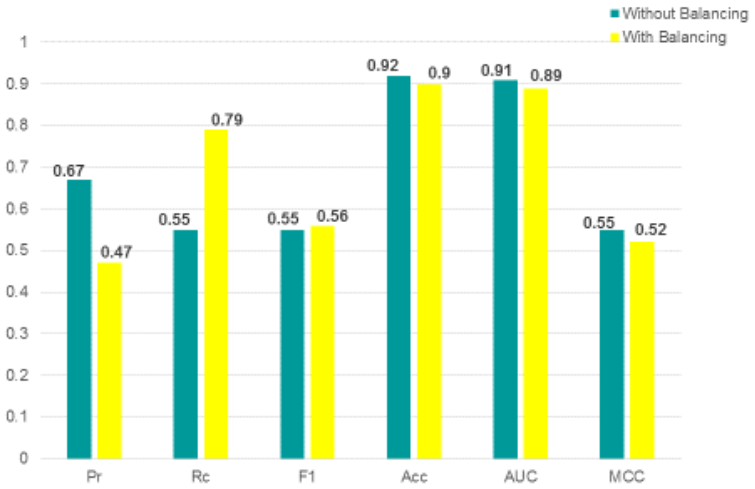
Conclusions

TEDIOuS

(TEchnical Debt IdentificatiOn System)



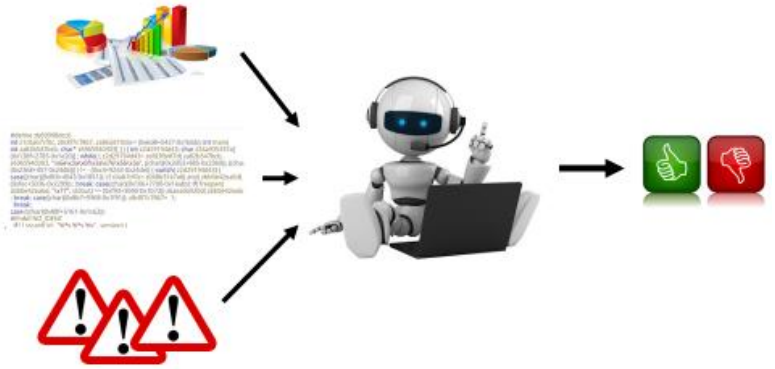
RQ_2: Performance across-project



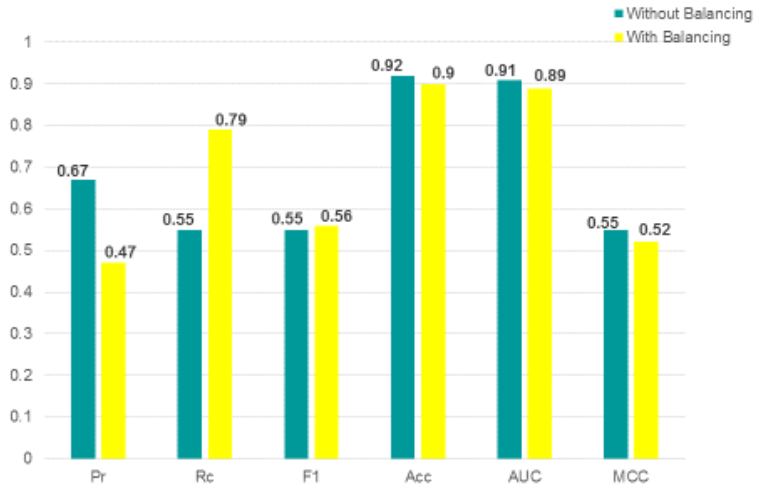
Conclusions

TEDIOuS

(TEchnical Debt IdentificatiOn System)



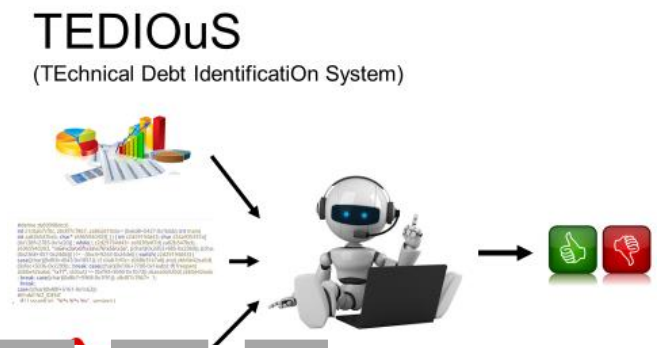
RQ_2: Performance across-project



RQ_1 & RQ_2: Performance within/across project

"Working at method level there is a relation between the internal quality and the number of Design SATD instances in it."

Conclusions



Questions and/or Comments
fiorella.zampetti@unisannio.it
@fio_zampetti