Universidade Federal da Bahia
Universidade Salvador
Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Identifying Technical Debt through Code Comment Analysis

Mário André de Freitas Farias

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

Salvador
25 de agosto de 2017

PMCC-Dsc-0029

MÁRIO ANDRÉ DE FREITAS FARIAS

IDENTIFYING TECHNICAL DEBT THROUGH CODE COMMENT ANALYSIS

Esta Tese de Doutorado foi apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Manoel Gomes de Mendonça Neto
Co-orientador: Rodrigo Oliveira Spínola

Salvador
25 de agosto de 2017

Ficha catalográfica.

Farias, Mário André de Freitas

Identifying Technical Debt through Code Comment Analysis/ Mário André de Freitas Farias– Salvador, 25 de agosto de 2017.

211p.: il.

Orientador: Manoel Gomes de Mendonça Neto.
Co-orientador: Rodrigo Oliveira Spínola.
Tese (doutorado)– UNIVERSIDADE FEDERAL DA BAHIA, INSTITUTO DE MATEMÁTICA, 25 de agosto de 2017.

1. Dívida Técnica. 2. Análise de comentários de código. 3. Engenharia de Software.
I. Mendonça, Manoel Gomes. II. .
III. UNIVERSIDADE FEDERAL DA BAHIA. INSTITUTO DE MATEMÁTICA. IV. Título.

CDU - XXX.YY

TERMO DE APROVAÇÃO

MÁRIO ANDRÉ DE FREITAS FARIAS

IDENTIFYING TECHNICAL DEBT THROUGH CODE COMMENT ANALYSIS

Esta Tese de Doutorado foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador.

Salvador, 25 de agosto de 2017

Prof. Manoel G. de Mendonça Neto (orientador), Ph.D.
Universidade Federal da Bahia - UFBA

Prof. Marcos Kalinowski, D.Sc.
Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio

Prof. Márcio de Oliveira Barros, D.Sc.
Universidade Federal do Estado do Rio de Janeiro - UNIRIO

Prof. Ivan do Carmo Machado, D.Sc.
Universidade Federal da Bahia - UFBA

Prof. Rodrigo Rocha Gomes e Souza, D.Sc.
Universidade Federal da Bahia - UFBA

ACKNOWLEDGEMENTS

(**Only in Portuguese**)

Antes de tudo, agradeço ao nosso Deus por todos os dias dedicados ao doutorado. Pela força e motivação para enfrentar as barreiras ao longo desses anos de plena dedicação e aprendizado. Agradeço a minha amada esposa, Janete Cahet, por todo apoio e paciência. Sem seu amor e presença diária, este trabalho teria sido muito mais árduo. Agradeço por nosso filho, Anthony, que me trouxe mais força e motivação no final desta jornada.

Agradeço imensamente a minha mãe (em memória) por toda minha formação básica. Por plantar em mim o amor pela docência e ensinar-me valores imensuráveis. Ao meu pai por mostrar que precisamos ser fortes e guerreiros e que a dedicação é um dos pilares do sucesso. Agradeço a ele por acreditar que a educação é um dos maiores tesouros desta vida. Agradeço aos meus irmãos e irmãs por toda inspiração e apoio. Aprendi com eles que por mais dura e triste que a vida possa ser, sempre temos que sorrir. Cada um de maneira individual é responsável por um pedaço deste doutorado.

Ao meu orientador Manoel Mendonça e ao meu co-orientador Rodrigo Spínola por todo o conhecimento compartilhado durante esses anos de pesquisa. Suas orientações foram além do arcabouço técnico e científico. Orientaram-me para uma vida científica justa e ética, sem egoísmos e elação. Foram realmente mestres na minha formação.

Ao amigo Methanias Colaço por ter contribuído fortemente na minha formação básica em computação, pelo incentivo na carreira acadêmica e científica.

Agradeço a Universidade Federal da Bahia e a CEAPG (seu corpo docente e técnico) por todo conhecimento compartilhado, apesar de todas as adversidades enfrentadas pelos profissionais de educação no Brasil. Aos colegas e amigos do doutorado (principalmente a Renato, Carol, Crescencio, Iuri, Raphael, Simone, Tássio, Thiago Souto, Gláucia, Magnavita e Thiago Mota) pela partilha de conhecimentos e por tornar os cafezinhos na cantina e no INES mais agradáveis. Aos colegas do grupo SoftVis por todas as discussões e crescimento científico. Gostaria de agradecer em especial ao amigo Amâncio Santos pelo companheirismo e pela considerável ajuda nas discussões da análise dos dados dos últimos experimentos desta tese. Agradeço, também individualmente, ao amigo Jaziel Lobo por todas as viagens de ida e vinda à cidade de Salvador, por ter sido um ótimo amigo de república e por dividir os mesmos sonhos acadêmicos. A Alcemir por toda troca de cohecimento durante as disciplinas do programa e durante esta pesquisa.

Por fim, agradeço ao Instituto Federal de Sergipe por proporcionar um ambiente de plena dedicação à minha formação no Doutorado em Ciência da Computação e aos colegas e amigos pelo incentivo e apoio. Em especial a José Osman, José Espínola, Ricardo Monteiro, Cristine, Glauco, Jean Louis, Almerindo Rehem, Jose Wlamir, Walter, Mauro e Railan Xisto. Aos amigos da vida Sergio Sampaio, Ricardo Fonseca, Wolney Sampaio, Dênio e Saulo Aragão.

ABSTRACT

Context: The identification of technical debt (TD) is an important step to effectively manage TD items and make TD manageable and explicit to keep the amount of TD under control. TD identification is the first TD management activity, and it is essential to know what types of TD exist, where they are, and what their impact on the project is. In this context, researchers have developed automated approaches to identify TD items using indicators derived from source code metrics. However, those indicators do not always point to TD that developer teams consider real problems and cannot identify many types of relevant TD. This work starts from the premise that one must employ several TD identification strategies to identify debts automatically. **Objective**: Our strategy is to consider code comments as an information source for TD. Code comments are readily available and store a richness of semantic information written in natural language. Our goal is to propose an approach to support and automate the identification of different TD types through code comment analysis. **Method**: First, we proposed a model and a contextualized vocabulary for identifying TD. Next, we develop a tool named eXcomment. The tool extracts and filters comments from source code using the vocabulary aiming to filter comments having a TD context. We then experimentally explore, evaluate and evolve the approach using a family of experiments named FindTD. **Work performed**: We evolve the approach over four studies. First, we performed an exploratory study (FindTD I) to characterize the feasibility of the proposed model. Next, we performed a controlled experiment (FindTD II) extending FindTD I with an additional quantitative analysis. Based on its insights, we run FindTD III changing the original setup and controlling other variables of interest. Finally, we performed the FindTD IV to show that is possible to identify automatically TD items through code comments using a contextualized vocabulary. **Results**: Our findings indicate that the model, vocabulary, and *eXcomment* make it possible to select a list of suitable comments to support TD identification automatically. The experimentation allowed us to evolve the model, vocabulary, and identify patterns that are useful to classify different types of debt. These experiments provided new evidence on how software engineers can use code comments to detect and classify TD items automatically. **Conclusion**: This thesis contributes to bridge the gap between the TD identification area and code comment analysis, successfully using code comments to detect several types of TD. The evidence indicates that exploring developers' point of view (human factors) can be useful to improve the current practice of automatic TD identification based of source code with more contextual and qualitative data.

Keywords: Technical debt identification, code comment analysis, contextualized vocabulary, software engineering

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CVM-TD	Context Vocabulary Model on Technical Debt
OSS	Open Source Systems
SE	Software Engineering
TD	Technical Debt
ESE	Experimental Software Engineering
SR	Systematic Review
MS	Systematic Mapping Study
LOC	Lines Of Code
GQM	Goal, Question, Metrics template
RQ	Research Questions
LIWC	Linguistic Inquiry and Word Count
NLP	Natural Language Processing
POS	Part-Of-Speech
MSR	Mining Software Repositories
ASA	Automatic Static Analysis
NP	Noun Phrases

Chapter 1

This chapter presents the motivation of this work, including problem statement, and the research goals and approach. Besides that, the chapter also discusses the methodology, contributions, and outlines this thesis

INTRODUCTION

The software industry often has to deal with several challenges to deliver and maintain products, such as providing useful software with high quality, on time, and on the budget. In practice, developing software aiming to put it only to work may be the easy part of the software development process. More than that, creating code that maintains the quality standards of the system is what stakeholders expect from our software engineers (PETERS, 2014). Unfortunately, development challenges lead developers to take shortcuts or use workarounds, helping them achieve short-term goals, without considering the possibility of they have a negative impact in the long-term, such as dirty design and poor implementation [2].

Creating a software product with a high level of quality under tough budget limitations is the main challenge of the software industry [3]. Frequently, this challenge is difficult, if not impossible, to overcome, and software engineers end up developing immature artifacts that cause unexpected delays and make the whole system difficult to maintain and evolve in the future. That is what the Software Engineering (SE) community now calls Technical Debts.

The term Technical Debt (TD) is a concept introduced by Cunningham (1992). Since then, it has been increasingly used to discuss technical compromises admitted by the development team during the phases of the software lifecycle. It refers to delayed tasks and immature artifacts that provide extra efforts in the future and cause "debts", increasing the cost of software evolution and maintenance [4].

The TD concept reflects the challenging decisions that developers and managers need to take to achieve short-term benefits to keep the customers satisfied and to survive in a competitive market. These decisions may not cause an immediate impact on the software, but may negatively affect the long-term health of a software system or its maintenance effort in the future [5]. For example, a developer when maintaining or evolving source code may choose the strategy of not updating the documentation about the changes to speed

up product releasing, but this decision may increase the complexity and maintenance cost in the long-term.

Debts can be associated to any immature software artifact. Some examples of immature artifacts are issues in the software design, incomplete or insufficient documentation, incomplete design specification, insufficient code comment, lack of adequate testing, overly complex code that needs to be refactored, or inadequate technology [6]. Furthermore, one may virtually insert TD during any step of the software development process and a TD can be associated with more than one software artifact.

The term "TD item" represents an instance of TD for the purpose of this thesis. A TD item may incur for different reasons: (i) **unintentionally**, when a developer may write a low-quality code due to lack of experience; (ii) **strategically**, when the project manager chooses to delay a maintenance task or performs it quickly with less quality in order to release the software earlier; or (iii) **limited resources and short deadlines**, when a developer insert a violation of modularity as a result of a short deadline to implement new software features.

In summary, the TD concept describes a situation where developers focus on tasks that can have immediate positive impacts on the software project and does not consider, or choose to accept, long-term maintenance costs into the decision making process [7]. The consequences of TD items can affect the value of the software, the costs of future maintenance, the schedule, and the software quality [4]. Consequently, a manager needs to balance short deadlines with long-term sustainability of the software when dealing with TD [8] [9].

1.1 MOTIVATION

The existence of TD items in software projects is inevitable, common, and can degrade the quality of software [10]. However, keeping the set of TD items under control is necessary to ensure productivity and to maintain the software project in time to market. On the other hand, TD items are not always visible and easy to identify. In general, a TD item is only known to few people from the development team and it is not visible enough to others who eventually have to pay for it (e.g., only a developer knows that a module has a documentation out of date) [11].

There are several challenges to manage TD items effectively, two important ones are: (i) identifying the set of TD items in the project before prioritizing them and select those that should be either paid or not, and (ii) identifying TD items that are really considered important by developer teams to reduce the maintenance and evolution cost of the software project. To this end, the TD identification is the first TD management activity. This activity is essential to know what types of TD items exist in a project, where they are, and to estimate the TD impact on the project [12].

In this research, we focus on the development of a TD identification approach aiming to complement the existing code-based TD identification strategies, for example, automated static analysis (ASA) and identification of code smells.

1.2 PROBLEM STATEMENT

Recent study [13] reported that the majority of works on automated TD identification focus on using static code quality analysis. Automated analysis tools have used software metrics extracted from the source code to identify TD items by comparing values of software metrics to predefined thresholds [14]. According to [13], the frequent use of source code explains the number of automated tools that analyze source code aiming to support the detection of TD and the existence of a high concentration of studies on types of debt more related to the source code, for example, code and design debts.

There are different indicators that have been used to support the automatic identification of TD items from source code. Automatic static analysis (ASA) and code smells are indicators that have been most frequently used [6]. ASA approaches analyze source or compiled code in order to look for violations of recommended programming practices ("issues"). These issues might degrade some dimensions of software quality (e.g., maintainability, efficiency). Some ASA issues can be used to indicate a part of code that needs to be refactored to avoid future problems [5]. Whereas code smells refer to potential violations of good Object-Oriented design principles in the source code and can be identified by comparing values of software metrics to defined thresholds [15].

The problem statement explored in this thesis is that in spite of these indicators being used to examine the relationship between software code quality, they do not always point to TD items that are considered a problem by developer teams or do not indicate to pieces of code which are regarded as a TD by developers. Moreover, some types of debt cannot be identified by source code metrics, for example, documentation debt.

This thesis starts from the premise that other automatic TD identification strategies can complement the code-based strategies, particularly those that can identify non-code-related debts that remain hidden in the software and may bring significant impact to the project in the future. A qualitative-based approach can certainly complement the traditional quantitative methods currently used to identify TD items.

In this sense, code comments are software artifacts which store a richness of semantic information written in natural language by developers who know the project (qualitative data) and may help us identify different situations of TD through a more qualitative analysis. Besides, these descriptions give us human readability that may trace the developers' point of view on required work in the future, and notice emerging problems, over which decisions need to be made [16].

However, to create automatic methods for comment analysis is a hard task because comments comprise natural language and have no mandatory format aside from syntactic delimiters. Hence, algorithmic solutions should be heuristic in nature [17]. In fact, research works in SE have widely been conducted with the focus on quantitative analysis.

This thesis investigates how code comments analysis can complement the existing TD identification with more contextual and qualitative data. It faces the problem of using qualitative methods to analyze comments to access the developers' point of view on what is a TD items, suggesting debts to which quantitative methods would be blind.

1.3 OBJECTIVES

This work seeks to expand the knowledge frontier on automated TD identification. Our objective is to develop an approach to automatically identify TD items and classify them into different TD types through code comment analysis. With this, we intend to complement the traditional code-based approaches with the qualitative data contained in code comments. This will allow us to identify different types of TD from different points of view. In order to achieve our primary objective, we derive four complementary goals described as follows:

1. Understanding how contextualized patterns of code comments may be categorized and combined to support the identification of different TD types;

2. Create a structure that systematically allows combining patterns providing a large vocabulary to support the TD identification;

3. Develop an automated tool to analyze developer's comments embedded in source code;

4. Evaluate and evolve the approach to identify TD through code comments analysis.

1.4 OVERVIEW OF THE METHODOLOGY

To achieve our objectives, we followed the research strategy represented in Figure 1.1. First, we proposed an initial model and contextualized vocabulary aiming to select comments possibly related to TD. Next, we developed a tool named *eXcomment*. The tool extracts, filters, and selects comments from source code using the vocabulary. Lastly, we explored, evaluated and incrementally evolved this approach through a family of four experimental studies (*FindTD*). Two of them had the purpose of characterizing and improving our strategy to identify TD items (*FindTD I and FindTD III*), whereas the others had the purpose of evaluating our approach (*FindTD II and FindTD IV*).

The four experiments produced five releases of the vocabulary and two releases of *eXcomment*. The first experiment (*FindTD I*) used the first release of the vocabulary which was generated from the proposed model. We used the results of *FindTD I* to evolve the vocabulary for *FindTD II*. Likewise, we used the results of *FindTD II* to evolve the vocabulary for *FindTD III*. We then evolved the *eXcomment* tool and used the results of *FindTD III* to evolve the vocabulary for *FindTD IV*. Lastly, the fourth experiment generated the fifth release of the vocabulary.

We focused on qualitative and quantitative analysis to synthesize the evidence in order to summarize, integrate, combine, and compare the findings of the set of experiments [18].

To combine quantitative and qualitative approaches as complementary methods, we used a triangulation methodology to analyze how code comment analysis approach supports the TD identification. Triangulation is a research strategy described as a convergent method with multiple operationalisms [19]. The main idea is to analyze evidence from different sources, to be collected and analyzed using different methods, have various forms,

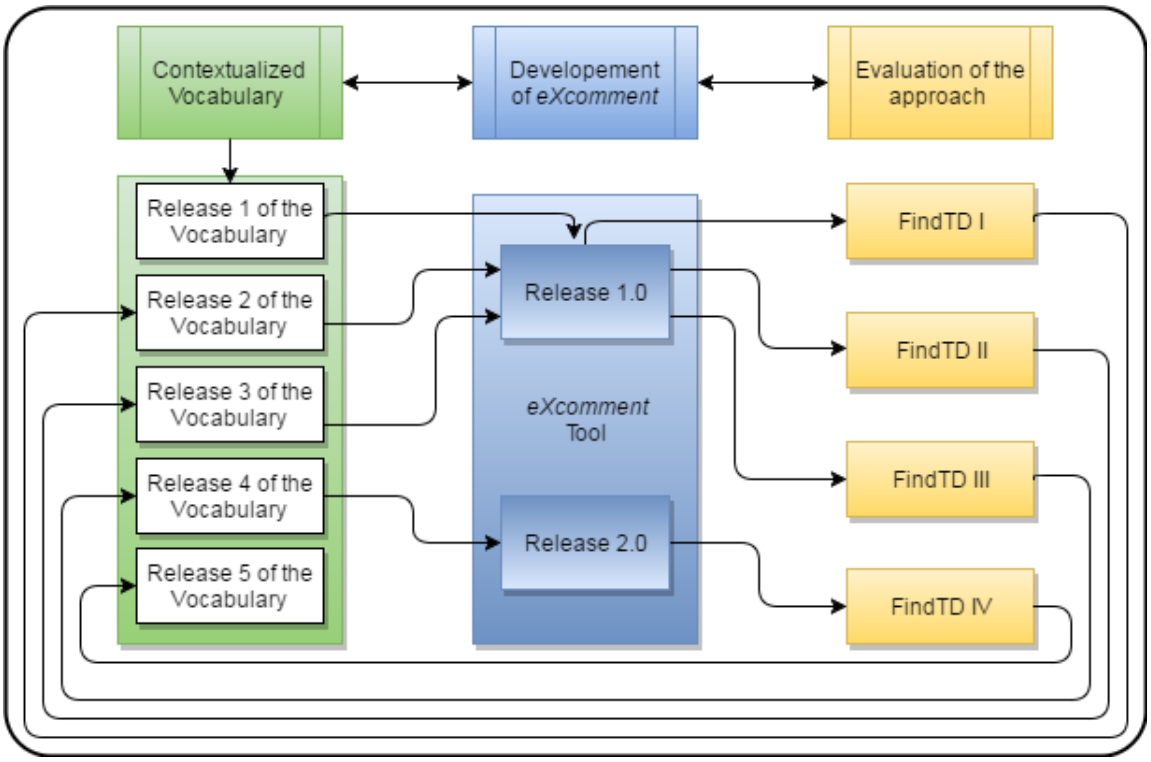


Figure 1.1 Empirical strategy.

or come from a different study altogether [20]. In this methodology, researchers can improve conclusions on their judgments through collecting and analysis of data considering the same phenomenon [21].

Figure 1.2 presents a different overview of the research methodology. The boxes represent the actions related to our goal. The boxes with dashed lines represent the tasks performed to create and evolve our model, vocabulary, and tool (tasks indicated with numbers 1, 2, 5, 7, and 8), and the boxes with rounded corners represent the empirical studies of the family of experiments (tasks indicated with numbers 3, 4, 6, and 9). We explored code comments using two systematic mapping studies, and the family of experiments in order to propose and evolve methods and techniques to support the identification of different types of TD. We briefly describe each step following the numbers of Figure 1.2.

Literature Review: initially, we performed a systematic mapping study (1) to understand the mining software repositories area and to identify its current targets and gaps, regarding mainly source code and comments analysis. We identified some relevant studies on the usage of comments for software comprehension, and some techniques and tools used to extract and analyze comments [22]. We also carried out a systematic mapping study on comment analysis (8). The primary objective of the study was to understand the state-of-the-art on comments analysis, investigating research enactment considering mining techniques, repositories, and tools used to extract and analyze comments. It identified techniques and algorithms to improve the *eXcomment*'s performance

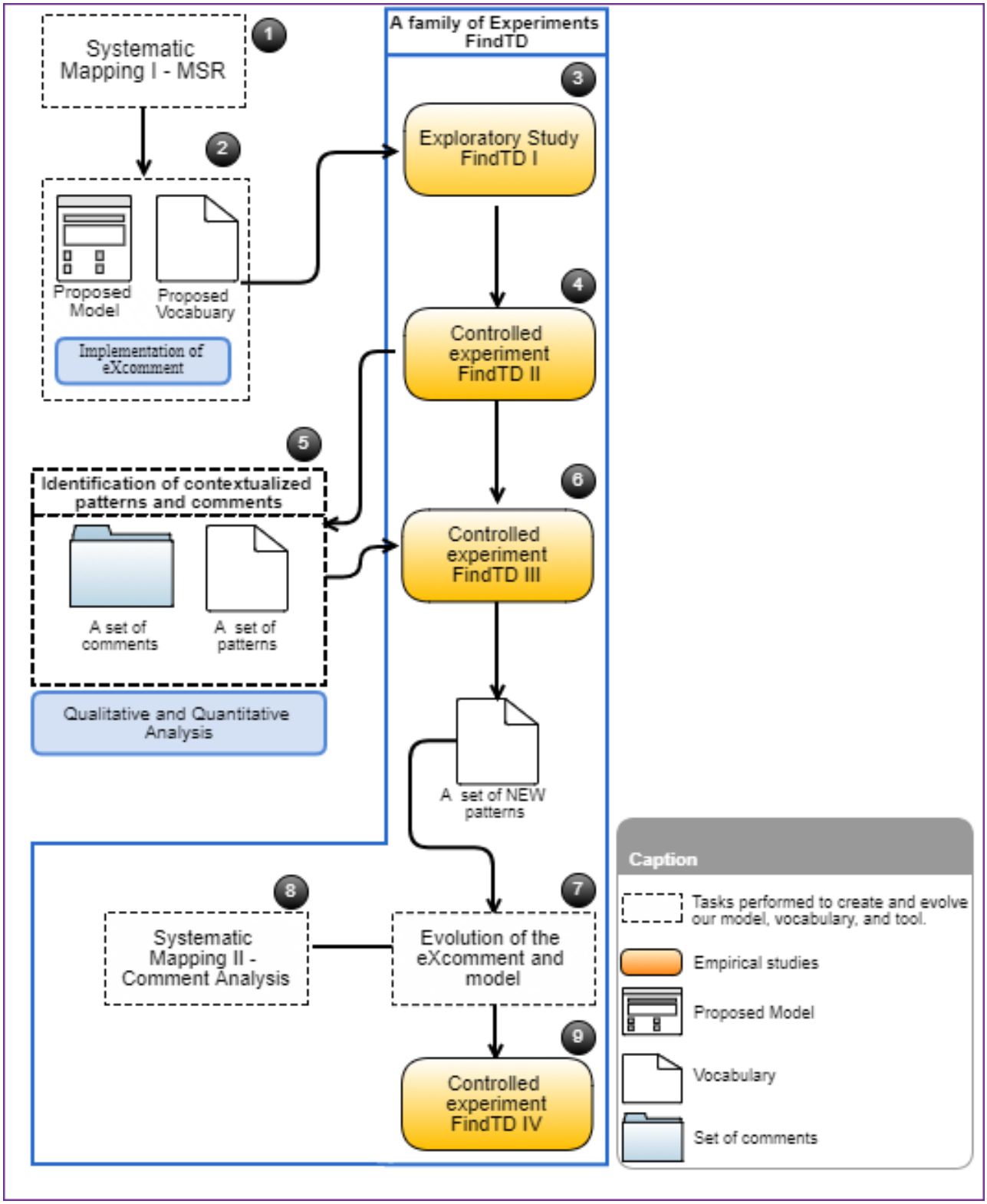


Figure 1.2 Overview of the research.

and searches.

Contextualized Vocabulary Model for identifying TD on code comments (CVM-TD): the model is a contextualized structure of patterns that focuses on using of word classes and code tags to provide a contextualized vocabulary on TD (2). CVM-TD systematically allows combining terms creating a large set of patterns on TD (vocabulary), for example, to match an adjective (the term "complex") and a noun (the term "method") creating a composed pattern "complex method". This task is related to second and third complementary goals.

eXcomment: to be able to extract comments from the source code, we developed the *eXcomment* tool (2). The first release extracts, filters, analyzes, and selects candidate comments from source code using the contextualized vocabulary provided by the model. However, the first release implements the selection of candidate comments in a semi-automatic way. The *FindTD III* provided us inputs to improve the *eXcomment* and the CVM-TD, resulting a new release of the tool and the contextualized vocabulary. Thus, we developed new features in *eXcomment*, creating the second release of *eXcomment*. The new release implements the searching of comments in an automatic way. To do that, *eXcomment* uses techniques from text mining such as preprocessing, tokenizing the unstructured text, and extracting terms. We detail the model and tool in Chapter 3. These features are associated with the new vocabulary to support faster interpretation of comments (7).

Family of experiments: our work ran a family of experiments called *FindTD*. A family of experiments involves not only replications but also variations among the experiments [23]. *FindTD* is composed of one exploratory study, and three controlled experiments. Figure 1.3 presents each experiment, their main objective, and their main results. The main purpose of the experiments is characterizing, evaluating and evolving our approach. First, we present *FindTD I* which was performed as an exploratory study. Next, we describe *FindTD II, FindTD III*, and *FinTD IV*, three controlled experiments.

The first experiment (*FindTD I*) (3) was performed to characterize the feasibility of the proposed model to support the detection of TD through code comments analysis. The results showed that the CVM-TD provides a vocabulary that makes it possible to extract comments that can be used to support TD identification [24].

Following, the promising initial outcome motivated us to evaluate CVM-TD with other data sources. Thus, we performed a controlled experiment (*FindTD II*) (4) analyzing the overall accuracy of CVM-TD when classifying candidate comments and factors that influence the identification of TD [25] [26].

Next, we performed the *FindTD III* (6) from insights of *FindTD II*, by changing the setup and controlling other variables. Our primary goal in this experiment was to analyze the set of comment patterns identified and classified in the previous experiment. We carried out a qualitative analysis to improve the model and the vocabulary, identifying the most significant patterns, and the relationship between comment patterns and TD types (5). The results provided us a new release of the contextualized vocabulary, improving our strategy to identify TD items.

Our last study was *FindTD IV* (9) The main goal of this study was to evaluate the whole process to identify and classify TD items through code comment analysis

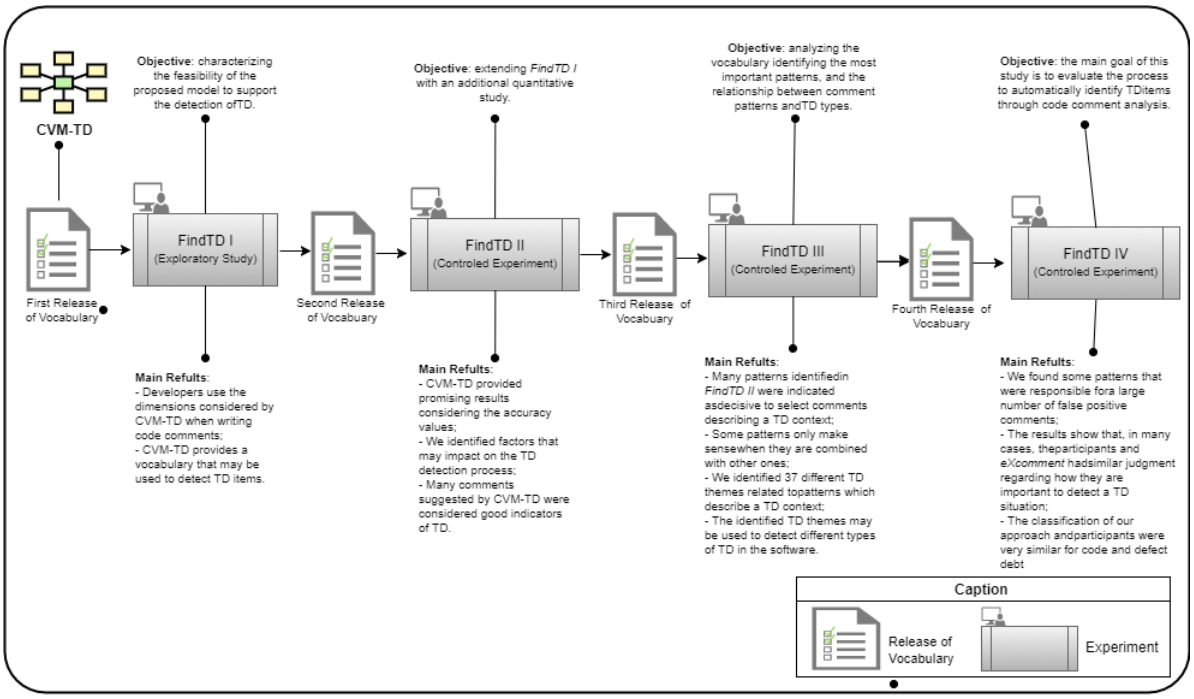


Figure 1.3 The empirical based research strategy.

automatically. We extracted, filtered, selected, and classified comments from two OSS projects using the fourth release of *eXcomment*. After that, we asked participants to analyze the candidate comments in order to point scores for each of them, classifying their type of TD.

We applied qualitative and quantitative data analysis over *FindTD IV* outputs to answer our research questions, aiming to evaluate and evolve once more our strategy to improve the detection and classification of TD items through code comment analysis.

1.5 CONTRIBUTIONS OF THE THESIS

The main contribution of this thesis is that it enhances the understanding on how comments can be used to identify and classify TD items. Currently, most works have focused on tools and methods for automatic detection of TD using code metrics. Research has mostly neglected the use of human-based qualitative inputs such as code comments. Through code comments, this research allows the use of qualitative information to automatically identify and classify TD items.

The proposed model, vocabulary, heuristics, and search strategies are relevant contributions because they can improve the automatic identification and management of TD items.

The empirical studies themselves are another contribution of this work. To the best of our knowledge, there is no other family of experiments focused on a large number of variables to evaluate a strategy to identify and classify TD items. These empirical studies

present interesting and complementary findings.

The results of the studies provide evidence and motivation for continuing to explore code comments in the context of TD identification. Our findings indicate that the model and vocabulary make it possible to extract a list of useful comments (those that were intentionally written by developers) to support TD identification. They also provide results that show that the use of the proposed strategy presented significant gains concerning accuracy values when the participants analyze code comments to detect TD.

The evidence shows that identifying TD through code comment analysis can be a hard task. However, our approach can be used by developers to detect automatically TD items in some cases, classifying them into some main types of TD. Also, even when that is not possible, it provides support to facilitate the analysis of comments by developers in order to detect TD items.

We made public the contextualized vocabulary and the dataset of comments extracted from the analyzed projects [27–30]. With them, other researchers and practitioners can advance works in TD identification area.

1.6 ORGANIZATION OF THIS THESIS

We organize the rest of this document as follows. Chapter 2 presents background work and important concepts associated with this thesis. Firstly, it discusses TD concepts and the text-mining field. Secondly, it discusses empirical studies on comment analysis applied to software comprehension. At the end, it discusses important references related to the use of code comments to identify TD.

Chapter 3 describes our approach to identify TD through code comment analysis. The chapter also presents the CVM-TD and eXcomment, a tool to extract, process, search patterns, and analyze code comments with the aim of identifying and classifying TD items.

Chapters 4, 5, 6, and 7 present the studies of the family of experiments used to characterize, evaluate, and evolve our approach. The chapters describe the planning, setup, execution, and threats to the validity of each study. The last sections of these chapters discuss the experimental results in detail.

Lastly, Chapter 8 presents an overall discussion of the research, summarizing its main ideas and contributions, exploring the main insights, outlining limitations and future works in the area.

Chapter 2

This Chapter presents a general view of concepts and works related to our study. First, Section 2.1 discusses conceptual aspects of TD. Next, Section 2.1.2 provides a brief history of the concept of TD and an overview of the TD landscape, discussing the main findings of two recent literature systematic mapping studies. In the second part of the Chapter, Section 2.2 discusses concepts, steps, and the process of text mining and Section 2.3 describes MSR concepts and works in this area. Finally, last Section talks about code comment mining and works related to software comprehension through code comments analysis, and it addresses studies directly related to this thesis them, the use of code comments to identify TD.

BACKGROUND

2.1 TECHNICAL DEBT

2.1.1 A Brief History of the Technical Debt Concept

In **1992**, Ward Cunningham described TD as writing immature or "not quite right" code in order to ship a new product to market faster [31]. This concept has been extended to refer to trade-offs between tasks to achieve short-term benefits, such as features to implement and time-to-market, and the long-term consequences to maintain and improve the system in the future.

Although the first concept is dated back 1992; it is only from 2010 that the studies on TD and its concepts have reached wider attention. One of the reasons is the appearance of the Managing Technical Debt (MTD) workshop as the primary venue for publishing and discussing TD in the SE community. At present, MTD is the leading conference in the area, disseminating results, new concepts, processes, methods, and tools in this field.

In **2010**, [32] defined TD as the debt that a development team incurs when it takes shortcuts in the software development process but that may increase software complexity and maintenance cost in the long-term.

In **2011**, [33] defined TD as any gap within the technology infrastructure or its implementation which has a material impact on the required level of quality.

In **2012**, [5] reported that the term TD is increasingly used to discuss technical compromises admitted by the development team during the phases of the software lifecycle. Thus, this metaphor defines the trade-off between internal tasks you choose to not perform at present and the risk of causing future problems. The TD metaphor reflects

the challenging decisions that developers and managers need to take in order to achieve short-term benefits to keep the customers satisfied and to survive in a competitive market. These decisions may not cause an immediate impact on the software, but may negatively affect the long-term health of a software system or maintenance effort in the future. Also, [34] argued that the concept of TD in software development has become somewhat diluted because various people have used the metaphor of technical "debt" to describe many other kinds of debts or ills of software development.

Zazworka et al. (2013) highlighted, in **2013**, that the concept of TD facilitates discussion among practitioners and researchers by providing a familiar framework and vocabulary from the financial domain and has potential to become a truly universal language for communicating technical trade-offs [35].

In **2014**, [6] discussed the coverage of concept of TD, which includes immature software artifacts such as issues in the software design and in the software architecture, incomplete or insufficient documentation, incomplete design specification, insufficient code comment, lack of adequate testing, and inadequate technology. In **2015**, considering the need to investigate the combination of these several definitions of TD and its application in practice, [36] conducted a two-part study to understand how software engineers relate to TD and whether they use tools and techniques to manage it. The results confirmed that the concept of TD has a broadly shared understanding among software practitioners and managers. In this sense, its acceptance and use have increased in software engineering research.

In this sense, [37] performed a systematic mapping study of the literature for the purpose of identifying decision-making criteria that have been used to support the management of TD. The study identified 14 decision-making criteria that can be utilized by development teams to prioritize the payment of TD items and a list of types of debt related to the criteria.

In **2016**, [13] described a TD as the debt that is caused by the development when it opts for an easy or quick solution to be implemented in the short term but with a high possibility of a negative long-term impact.

Lastly, in **2017**, [2] complete the other concepts stating that TD is a term being used to express non-optimal solutions, such as hacks and workarounds, that are applied during the software development process.

All of the concepts share core information about TD, which is the Trade-off between internal tasks you choose not to perform at present and the risk of causing future problems. For this reason, this thesis will adopt the definition stated by [7]. According to them, ***the TD metaphor describes a situation where developers focus on tasks that can immediate impacts on the software project and does not consider long-term maintenance into the decision making process***.

2.1.2 An Overview of Technical Debt Landscape

To present an overview of TD landscape, we discuss the main findings of two literature systematic mapping studies, [12] and [13], which present, to the best of our knowledge, the current TD landscape. Through the mappings, we intend to provide a comprehensive

view about the current TD research.

The primary goal of the systematic mapping performed by [12] was to take a comprehensive understanding of the concept of TD and provide an overview of the current State-of-the-art on TD management. To this end, they collected studies on TD and TD management, making a classification and thematic analysis on these studies.

Their results show that although the term TD had become widely used in software engineering, different approaches have been employed it in a variety of ways, which led to ambiguous interpretations. This finding is in line with the study by [34].

The study [12] classified TD into ten different types (Requirements, Architectural, Design, Code, Test, Build, Documentation, Infrastructure, Versioning, and Defect TD), and 8 TD management activities (TD repayment, identification, measurement, monitoring, periodization, communication, prevention, and representation/documentation). Among the ten types of TD, code TD is the most studied one. Test, architecture, design, documentation, and defect debt have also received significant attention.

The TD management activities have received different levels of attention. TD identification, repayment, and measurement have been studied more frequently, accounting for more than half of the total studies on TD management. TD identification is the activity presenting the most approaches among the analyzed studies.

They analyzed and classified 29 different tools. From this set, only one (FxCop) takes .NET assemblies as input, one (RE-KOMBINE) takes requirements and solutions as input, and one (CLIO) takes compiled binaries as input. Most of the tools (86%) take source code as input and focus on the TD identification activity. Some of these tools are: (i) SIG Software Analysis Toolkit, which calculates code properties to identify code TD [38], (ii) Resource Standard Metrics – which, estimates source code metrics and analyzes code quality to find style violations and logic problems to identify design and code TD, and (iii) CodeVizard, which detects design TD thought code smells identification [39].

In [13], the authors conducted a systematic mapping study on the TD field. They aimed to investigate strategies that have been proposed to identify and manage TD in the software lifecycle. The authors broke down the objective into five goals to: (i) characterize the types of technical debt, (ii) identify indicators that can be used to find technical debt, (iii) identify management strategies, (iv) understand the maturity level of each proposal, and (v) identify what visualization techniques have been proposed to support technical debt identification and management activities.

The study identified fifteen types of TD, ten of which are in common with [12]. The five new ones identified by them are Usability, Test Automation, Service, Process, and People debt. Hence, this classification complements the categorization proposed by the latter. Moreover, they identified and classified a list of indicators sorted by TD type. They refer TD indicators to software development activities, so that these indicators allow the discovery of a TD item by analyzing different artifacts created during those activities.

The list of TD management activities identified by [13] is more comprehensive, than the one presented by [12]. The former identified 20 different strategies. In total, the studies provided a complementary set of 28 TD management strategies.

Both reviews reported that code quality analysis techniques have frequently been

considered to support the TD identification. They both also highlighted the need for more empirical studies on TD evaluation and application of some approaches in practical settings. Alves et al. noticed that although many indicators have been characterized, few of them have been evaluated.

Regarding software visualization techniques, [12] dealt with this theme as TD communication approaches. According to them, some studies focused on dependency visualization and code metrics visualization as resources to make identified TD visible to stakeholders, so that it could be further discussed and managed. Whereas [13] analyzed software visualization in the context of TD identification.

2.2 TEXT MINING

Our work is based on text mining, a technology for analysis of large collections of unstructured data, aiming to extract patterns or interesting and non-trivial knowledge from code comments [40]. Similar to conventional data mining, text mining consists of phases that are inherent to knowledge discovery process [1]. These steps may vary from different authors and approaches, but most of them comprise at least data selection, preprocessing, and mining, as can be seen in Figure 2.1. According [1] and [41], the steps are summarized as:

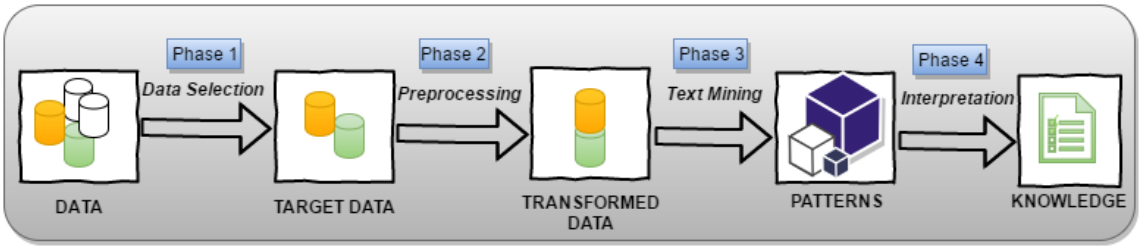


Figure 2.1 Overview of the steps constituting the knowledge discovery process [1].

1. **Data selection:** the main goal is creating a target dataset on which discovery is to be performed;

2. **Preprocessing**: the original documents are not always presented in a purely textual format. Therefore, it is necessary to convert them to a standardized format, eliminating any attributes of presentation formatting. This phase includes basic operations, such as removing noise or outliers if appropriate, and characters conversion to uppercase or lowercase. Selection of simple or compound words is a step of this phase. In some cases, during the preprocessing of a document, several common words (phrases) may be managed as a single term. This selection can be made using predefined word lists or statistical and syntactic techniques. Text mining pays particular attention to preprocessing because data is unstructured for computer analysis. In other words, after setting the base with texts to be mined, it is necessary to convert each document to a suitable format for a computational process;

3. **Data Mining**: includes selecting method(s) to be used for searching for patterns in the data, such as deciding which models and parameters may be appropriate. It also includes searching for patterns of interest in a particular representational form or a set of such representations, including classification rules or trees, regression, clustering, sequence modeling, dependency, and line analysis. This step consists of associating information to each term regarding its positioning and semantic for the analyzed scope;

4. **Interpretation**: includes interpreting the discovered patterns and possibly returning to any of the previous steps, as well as removing redundant or irrelevant patterns, and translating the useful ones into terms understandable by users. In this phase, possible visualization of the extracted patterns is used to comprehend the discovered patterns easily and transform them into knowledge. Finally, one should manually verify the text analysis (e.g., sampled), in addition to regular bias reporting [42].

The main difference between data and information retrieval is precisely the relevance of the obtained information [17]. In this sense, not all terms that compose a document are relevant when one intends to analyze different scopes. Thus, identifying words or expressions with high semantic content is important to select and filter those that are meaningful for the objectives in focus.

A vocabulary and/or thesaurus, representing previous knowledge, may provide an appropriate solution to extract and recognize parts of the text that are significant to the analysis. On the other hand, handling text data is a hard challenge because natural language is flexible, frequently appearing new expressions and roles. As a consequence, building a vocabulary covering all words, expressions, and roles is almost impossible [43].

Another strategy is N-Grama analysis. Ogada and Mwangi [44] defines an N-Gram as a simply sequence of consecutive symbols extracted from a long string. N-Gram is a tokenized text snippet and it is a commonly used technique in the field of computational linguistics to analyze a large text dataset with the purpose of identifying combinations of keywords which frequently appear together. N-gram feature analysis often goes beyond U-gram (one term) features to also include bigram (adjacent word pair) features (e.g., bad code) [45]. Note that 1-gram is simply the individual words N-Grams have been used in duplicate detection. In this work, we use N-grams of several different lengths simultaneously.

According to [46], it is possible to compute how important a certain keyword by analyzing N-Grams and performing a frequency analysis. However, N-Gram analysis does more than a simple frequency analysis because it takes into account how words are combined. For example, the occurrence of the words "complex", "code", "bad", and "smells" in the phrase "The following method is a complex code, it is a bad smell" might not indicate anything interesting, but once they are identified as N-Grams can signal an interesting clue, for example, "complex code" and "bad smells".

N-gram techniques have been successfully used for a long time in a wide variety of problems and domains [47]. For example, N-Gram has been used to mine and understand

papers [46], analyze comments and programming errors [48], and quantify security effort in the software development Lifecycle [45].

Another common task of the preprocessing phase is the removal of stopwords. Stopwords are terms that are not considered relevant in the analysis of text [43]. In general, a use of a word list to be ignored. This list consists of a relation of words that have no significative semantic content (e.g., prepositions, conjunctions, articles, numerals etc.) and consequently are not relevant for text analysis. In addition, special collections of documents may have extra stopwords, e.g., "ArgoUML" can be a stopword in a collection of commetns from ArgoUML project.

We developed a tool (*eXcomment*) which addresses all of these phases and steps to extract and analyze code comments. We describe how we applied them in Section 3.1.

2.2.1 Text Mining Techniques in Software Engineering Approaches

Text mining techniques have been used to explore several SE approaches. For example, to identify security bug reports [49], to predict the severity of a reported bug [50], to identify Open Source Software (OSS) developers' context-specific preferred representational systems [41], to analyze existing literature in order to perform a secondary study on mining software repositories [46], and more recently to categorize Stack Overflow questions based on their importance [51] and to combine text mining and visualization techniques to study team artifacts [52, 53]. Stack Overflow [1] is a question and answer (QA) site for programmers. We next present a brief description of each.

In [49], the authors identified security bug reports via text mining through an industrial case study. They developed a new approach that applies text mining on natural-language descriptions to identify security bug reports that are manually-mislabeled as not-security bug reports. Security engineers can use the model to automate the classification of bug reports from large bug databases aiming to reduce the time that they spend on searching for reports. In a similar context, [50] showed that it is possible to predict the severity based on other information contained in a bug report, in particular, the textual information describing the bug, using text mining algorithms. The work also investigated subsidiary research questions inherent to the utilization of a text mining algorithm, such as "Which terms and text fields in the textual descriptions of a bug report could serve as good indicators of the severity?".

In another perspective, text mining was used to identify OSS developers' context-specific preferred representational systems [41]. This work presents a psychometrically-based neurolinguistic method to determine the preferred representational cognitive system of software developers. For that, the authors developed a psychometrically-based neurolinguistic analysis tool, which uses a vocabulary and Linguistic Inquiry and Word Count (LIWC). The tool combines text mining and statistical analysis techniques in order to classify programmers.

The study [46] presented an analysis on existing literature on mining software repositories. They applied a text mining approach on the complete corpus of the Workshop on Mining Software Repositories papers to investigate how the research field has evolved in

[1] http://stackoverflow.com

the last decade. It was a novel text mining approach to understand the trends on the research topics and how they are referred to in the MSR literature. The authors converted pdf files into text format. On these text files, they performed a series of post-processing steps and generate N-Grams analysis. They address issues like the trendy research topics; the frequently (and less frequently) cited cases; and the popular (and emerging) mining infrastructure.

More recently, text mining has been used to analyze discussion forums. The work [51] has as overall goal to understand the common challenges and/or misconceptions among web developers. To that end, authors extracted and mined more than 500,000 Stack Overflow questions related to web development containing the following three tags, namely, JavaScript, HTML5, and CSS.

Finally, visualization may also supplement text mining to reveal unique multi-dimensional insights. The objective of [52] was to demonstrate the utility of combining these approaches to improve the comprehension of extracted data and facilitate human understanding. Visualization techniques may be combined with text mining to enable the SE community to analyze detailed information about team artifacts. This type of approach is not directly related to our primary objective, but it is relevant because we intend to adopt visual paradigms in our study.

In [53] and [54], the main idea was to mine two repositories of the Apache Httpd project[2] to gather information about its developers' behavior. An approach to cross data collected from a mailing list and the project source code repository through mining techniques was developed. The approaches use visualization techniques to analyze the mined data.

2.3 MINING SOFTWARE REPOSITORIES

The Mining Software Repositories (MSR) field focuses on uncovering interesting and useful information about software projects and analyzing available data from different software repositories. Using the information stored in these repositories, practitioners become less dependent on their intuition and experience, and more dependent on historical and field data [55].

Software repositories contain a large amount of software historical data that can include valuable information on the source code, defects, and other issues like new features. Through mining repositories, practitioners can uncover useful and important pattern and hidden information. Thereby, MSR has become an important area for SE researches and a popular tool for empirical studies in this area [56].

Several researches have been exploring MSR, aiming to explore and discuss topics such as prediction of defects [57–59] and comprehension of software evolution [60–62].

In general, performing MSR studies is a challenge for researchers since they need to deal with several SE artifacts, data sources, techniques and understanding how better evaluate their purposes and results. In order to understand the MSR area and to identify its current targets, shortcomings, and gaps, we performed a systematic mapping study over studies published at five years of MSRConf (from 2010 to 2014). We chose this fresh

[2]http://http://apache.org/

period in order to investigate recent studies on MSR area. This task was performed at the beginning of this thesis and from that, we defined the main goals we explored in this work.

We have extracted and analyzed data from 107 papers. Our findings show some trends on current use of purpose, focus, object of analysis, source data, and evaluation methods at MSRConf. This allowed us to investigate how researches are being conducted in this field.

We have also identified some gaps with respect to their goals, focus, and data source type (e.g. lack of usage of comments to identify smells, refactoring, issues of software quality, and TD). Regarding the evaluation methods of the studies, our analysis pointed out to an extensive usage of some types of empirical evaluation. The main findings and insights are described below and the whole mapping is presented in Appendix A.

A large set of purposes and focus have been used for many different goals. The main goals are comprehension of defects, analysis of contribution and behavior of developers, and software evolution comprehension. "Code" was the most used object of analysis in this field.

We defined a taxonomy that divided the data source types into structured and unstructured repositories. From this analysis, we might identify the main data source types in each category. Structured is more explored than unstructured repositories, but the number of approaches using unstructured data source is increasing in the last three years. Other software engineering artifacts are coming to be used, such as comments, emails, and microblogging. They have been analyzed alone or together with metrics extracted from structured repositories to understand quality issues in software projects, for example, comments and code metrics.

Considering evaluation methods, we identified that almost all studies have performed some type of empirical evaluation and few number of works has conducted "controlled experiments" and "surveys". Furthermore, another contribution of our study is that we make available the dataset of our data extraction and our classification [63].

Having a broad view of the area considering recently published studies helped us know the MSR community and explore works using different purposes, focuses, objects of analysis, or data sources. Thus, this mapping showed us some gaps identified in the MSR area. This thesis is interested in the gap between code comment analysis and TD identification. This encouraged us to start investigating characteristics of software that may indicate a quality problem in the software development, analyzing the code comments domain in order to design and evaluate new methods and tools to support TD identification.

From this study, we performed a new mapping study focused on code comment analysis that is presented in the next section. Although some systematic literature reviews have been performed in the MSR area [42, 46, 64], none of them has focused on comments analysis as object of study. In this context, the goal was to investigate how comments analysis has been explored with the purpose of supporting software engineering activities.

2.4 CODE COMMENTS MINING

Comments are a generic type of task annotation (short descriptions) where developers insert documentation directly into source code expressing their intentions and assumptions [65]. These annotations are a richness of semantic information written in natural language and help developers to understand the code for future maintenance or reuse.

In accordance with [66], code comments and the source code itself are an important documentation to help understand a system. They are important software artifacts which may help to understand software features [65]. Code comments mining may reveal valuable information such as the reason for adding new lines to the source code, knowing about the progress of a collective task, or even why relevant changes were performed or not. Thus, comments may be used to discover issues that may require work in the future, notice emerging problems and which decisions need to be taken about them [67]. In summary, code comments may describe the developers' point of view on quality issues in the software development and can be explored to give us human readability and provide information that summarizes the developer context [16].

When the source code is well commented, we can understand what a piece of code does, what issues it has, and whether it needs to be fixed or improved, without needing to analyze its implementation. In general, comments are used by developers to understand unfamiliar software because comments are written in natural language [68]. Therefore, comments document the code and help developers understand the implementation, issues, and its solutions. So, source code comments are considered a necessary documentation that is used in maintenance. They complete the general documentation of a system and are a convenient way for developers keep documentation and code consistently up to date [17].

Regarding co-evolution of code comments and changes in source code, [69] showed that source code and code comments have a high co-evolution. The authors found that 97% of the comment changes are consistent with source code.

Despite there are different syntaxes and types of comments according to the programming language, they are divided into two categories: (i) inline comments, which only permit the insertion of one line of comment, and (ii) block comments, which allow the insertion of several lines. Developers write comments in a sublanguage of English using a limited set of verbs and tenses, and personal pronouns are almost not used [70].

According to [17], these categories can be differentiated between seven types of comments:

- **Copyright comments**: notes that include information about the copyright or the license of the source code file. They are usually found at the beginning of each file;
- **Header comments**: give an overview about the functionality of the class and provide information about the class author, the revision number, or the peer review status;
- **Member comments**: describe the functionality of a method/field, being located either before or in the same line as the member definition. This type of comment is similar to header comments;

- **Inline comments**: describe implementation decisions within a method body. This type was named as the same description of the category of comments presented above.
- **Section comments**: this comments address several methods/fields together belonging to the same functional aspect;
- **Code comments**: contain commented out code which is source code ignored by the compiler. Often code is temporarily commented out for debugging purposes or for potential later reuse;
- **Task comments**: are a developer note containing a todo statement, a note about a bug that needs to be fixed, or a remark about an implementation hack.

In this work, we are interested only in the inline comments and task comments types because they can describe the developer's feeling about open tasks in the project and the risk of causing problems if not done in the future.

However, in general, performing comment mining studies to analyze comments from code and identify quality issues in software projects is a challenge. We need to deal with different techniques from text mining such as preprocessing, tokenize the unstructured text, term extraction, Part-Of-Speech (POS) tagging, linguistic dictionary, and understanding how better evaluating their purposes and results.

To conduct a broad investigation of understanding the code comment mining area and to identify its current targets, shortcomings, and gaps, we extend the first mapping, cited in the previous subsection, and performed a second systematic mapping study focused on code comment mining. We focused on studies published at the leading digital libraries: ACM Digital Library, IEEE Xplorer, Science Direct, Engineering Village, Springer Link, Scopus, and Citeseer. This study was carried out together with the TD research team[3].

The primary goal was identifying which purposes, focus, techniques, and tools have been used in these studies to support SE area. We have extracted and analyzed data from 36 papers. Our findings show some trends on current use of purpose, focus, techniques, and tools have been used in current researches exploring code comment mining. This evidence allowed us to investigate how techniques and tools are being used to mine and analyze comments. The main findings and insights are described below, and the complete mapping is presented in Appendix B.

We identified that studies exploring comment analysis with the goal of supporting activities of SE are aligned with the purposes of MSR area presented above, mainly considering comprehension and identification purposes. We also identified that comments had been explored to detect and comprehend the quality of software artifacts and TD. It is important to highlight that the focus on TD started to be considered recently, in 2014.

Regarding techniques used to mine comments, we identified 14 different techniques that were explored by many studies. We observed that Dictionary/Vocabulary, NLP and Statistic Analysis were the most investigated techniques in the comment mining process, considering the analyzed studies. We also identified that 55.6% of the identified

[3]TD Research Team - http://www.tdresearchteam.com

techniques are semi-automatic, followed by automatic (27.8%), and manual techniques (8.3%). A possible reason for the low usage of manual techniques may be because it is hard to perform a manual analysis regarding effort and also due to the process being error-prone.

Most identified tools are targeted as the extraction step, and we noted that few tools were used to process and analyze comments. Another important point is that the tools do not cover all comment mining process so that we found only one tool which is used in more than one comment mining step (*iComment*). Thus, many tools were developed to address only a step of comment mining process. A possible explanation for that is that each study has a specific need of exploring comments in order to achieve its goals. For example, srcML[4] extracts comments from source code, but it does not filter them by dividing comments that were automatically generated by IDEs and those that were written by developers.

From this mapping, we identified some important studies on the usage of comments for software comprehension and some techniques and tools used to extract and analyze comments. Finally, the findings provided us with the knowledge and motivation to evolve our tool to be able to extract, process, and analyze code comments automatically in an interactive process.

2.4.1 Comment Analysis Approach for Software Comprehension

As earlier discussed in the previous Section, comments provide an important set of information which may help to understand software features, make software comprehension easier, and assist in task assignment, management, and completion. Code comments have been used as data source in some works to discuss and analyze their importance on software comprehension from different point of views.

Through both systematic mapping studies, we identified some important researches from MSR and code comment areas which have focused on code comments mining. The main works are [16, 65, 68, 71–74]. However, the most interesting discussion for us is related to usage of code comments to identify TD (we discuss them in the next section). Below we describe some of these MSR works in more detail.

In [16], the authors analyzed the purposes of work descriptions (task comments) aiming to discuss how automated tools can support developers in creating them. In the same line, [65] focused on the analysis of task comments to explore how code comments play a role in how developers deal with software maintenance tasks. They performed an empirical study by combining results from a survey of professional software developers, an analysis of code from open source projects, and interviews with software developers. The findings show how task annotations can be used to support a variety of activities fundamental to articulation work within software development and detailed how comments may improve processes and tools that are used for managing these tasks. Both studies are related directly to concepts addressed in our proposed model presented in next Chapter.

The study [68] present an approach to locate problem domain concepts on comments and identify the relevant code chunks associated with them. The authors also introduce

[4]http://www.srcml.org/

Darius, a tool to implement their proposal for Java programs. Darius identifies and extracts inline, block, and Javadoc comments and provides some metrics. In such approach, comments are isolated marking their category (inline, block or Javadoc comment) and keeping their context (code lines to which they are associated) but the tool does not distinguish comments from their types, for example, task comments or code comments. They concluded that higher level source entities tend to have comments oriented for problem domain information, whereas comments of lower level source entities tend to include more program area information.

In other work, [73] suggested a POS tagging of program identifiers to understand how a program element is named. Many text-based tools for SE use POS taggers, which identify POS of a word and tag it as a noun, verb, preposition, etc. to help distinguish the semantics of the component words. Nevertheless, discovering only the word classes into phrases cannot contribute to distinguish the semantics of the component words. The main task of the POS tagger is to assign part of speech tags to each word in the text.

Yang and Tan [72] proposed an approach that analyses the word context in code comments. The main idea is to discover semantically related words by analyzing the context of words in comments and code. Many words that are semantically related to software development process are not semantically related in English. In this same sense, Howard et al. [71] also presented an approach to augment natural language thesauri with Code-Related terms. Considering identifiers and comments, [74] performed an experiment with students and young professional developers in order to understand how they perceive comments and identifier names.

2.4.2 Using Code Comments to Identify Technical Debt

Code comments have been explored with the purpose of identifying TD [75] [76] [77]. Comments provide a perspective which allows considering more contextual and qualitative data, human factors and the developers' point of view in order to complement the quantitative analyses in the TD identification process. Developers use comments to describe situations where they know a current implementation that is not optimal and write comments alerting the inadequacy of the solution [76]. In other words, these task annotations may refer to situations of TD by describing immature software artifacts, internal tasks you chose to not perform at present, and the risk of requiring extra effort to correct and modify them in future.

2.4.2.1 Correlated studies. Since this thesis focuses on the identification of TD, we discuss the studies related directly to concepts addressed in this work, which analyze comments to detect TD items.

In [75], the authors read manually 101,762 code comments to identify those that point to a Self-Admitted TD. These comments were analyzed in order to determine the text patterns that indicate a TD. In total, 62 recurring patterns (e.g. hack, fixme, stupid, hope everything will work) were identified. According to the authors, these patterns can be used to manually detect TD items that exist in the project by reading code comments. For that, the authors used the srcML toolkit [78], a command line tool that

parses source code into an XML file, to extract the comments. In this step, the authors considered all types of comments. This decision may bring a lot of unnecessary effort because it considers comments that are not important to TD scopes, such as license and Auto-generated comments.

After the data extraction, the authors identified comments that indicate TD. Their findings show that 2.4 - 31.0% of the files in a project contain self-admitted TD. In addition, the most used text patterns were: (i) "is there a problem" with 36 instances, (ii) "hack" with 17 instances, and (iii) "fixme" with 761 instances.

In another TD identification approach, [76] evolved the work of [75] proposing four simple filtering heuristics to eliminate comments that are not likely to contain technical debt. To do so, they read 33K code comments from source code of five open source projects (Apache Ant, Apache Jmeter, ArgoUML, Columba, and JFreeChart). Their findings showed that Self-Admitted TD can be classified into five main types: design debt, defect debt, documentation debt, requirement debt, and test debt. According to the authors, the most common type of self-admitted TD is design debt (between 42% and 84% of the classified comments).

Another study focuses on TD identification through code comment analysis. [79] examine the relation between TD and software quality. This work concentrates on the identification and examination of TD. It is important to note that the authors performed their study on five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. To identify TD items, they followed the methodology applied in previously cited work [75], which uses patterns indicating the occurrence of TD. As the main result, the study brought empirical evidence on the fact that a set of TD items may impact the software development process, mainly by making it more complicated. Hence, practitioners need to manage it properly to reduce its consequences.

Bavota and Russo [80] also used a set of patterns to identify TD items. They presented a differentiated replication of the work presented by [75]. They carried out a large-scale empirical study across 159 software projects to investigate the diffusion and evolution of self-admitted TD and its relationship with software quality. Their main results are threefold. First, they showed the dissemination of TD instances (on average, 51 instances per software project). Second, they identified that, on average, the TD instances have a great survivability. Third, the set of TD items tends to increase over the projects lifetime. The results highlight the need of techniques and tools aimed at supporting the effective identification and management of the set of TD.

The study [77] conducted an investigation to reveal the different types of TD that can lead to the rejection of pull requests. The main goal was to explore the identification and classification of TD introduced by the developers through analysis of comments in pull requests. They classified TD into seven types of debt: design, documentation, test, build, project convention, performance, and security. The results also indicate that the most frequent TD is design (39.34%), followed by test (23.70%), project convention (15.64%), performance (9.48%), documentation (5.69%), build (2.37%), and security (1.42%). In the same way that the first work presented by [75], this work is hard to perform such manual analysis in terms of effort and also due to the process is error prone. There is no a semi-automated or automated resource to eliminate the pull requests that were not

interesting for the proposed analysis. Another point is that this strategy may involve different interpretations depending on who is reading the comments.

Next, [2] has used Natural Language Processing (NLP) to identify design and requirement debts automatically. The authors extracted and classified the code comments from ten OSS projects, by using 62 comment patterns derived from [75]. The results show that terms related to sloppy code or mediocre source code quality are the best indicators of comment to detect design debt, whereas terms related to the need to complete a partially implemented requirement in the future are the best indicators to identify a requirement debt. Moreover, the results show that developers use a rich vocabulary to express different TD situations.

More recently, [11] proposed an automatic approach to detect self-admitted technical debt in source code using text mining. The authors extended the work by [75], improving and automating their classifier. They combine multiple classifiers from different source projects to build a composite classifier that identifies TD comments in a target project. The new approach was evaluated on eight open source projects, using F1-score measure. The experimental results show that the new approach improves over the state-of-the-art method proposed by Potdar and Shihab by 499.19%. Their proposed approach can be used by project personnel to identify TD with minimal manual effort effectively. However, it still needs a manual intervention.

These research studies provide preliminary indication that comments can be used to support TD identification and they are in line with the purpose of this work. They were carried out concurrently with this research, and they are very correlated to our proposal. However, our approach broadens the patterns proposed by [75], since we focus on how terms [5] may be combined to identify different TD types. Besides, CVM-TD can easily be extended and automated to quickly analyze code comments.

[5] In this work, we call such terms patterns.

Chapter

3

This chapter presents our approach to identify technical debt through code comment analysis, including the description of a contextualized vocabulary model and the eXcomment tool. The chapter describes how we created the model and generated the contextualized vocabulary. It also details each step of the eXcomment, which is a tool to extract, process, search patterns, and analyze code comments with the aim of identifying and classifying technical debt items

IDENTIFYING TECHNICAL DEBT THROUGH CODE COMMENT ANALYSIS

In this Chapter, we describe the Contextualized Vocabulary Model for Identifying Technical Debt (CVM-TD). The model is a contextualized structure of patterns that focuses on the use of word classes and code tags to provide a TD vocabulary, aiming to support the detection of different types of debt through comment analysis.

To create CVM-TD, we have considered patterns that may indicate bad code quality, activities performed or not by developers, issues that may report a future work, and an ontology to identify software engineering nouns [81]. Besides, we also hypothesize that, in general, a composed pattern is more semantic than an isolated word since they bring with it some context information. For example, the pattern "stupid" found in a code comment does not bring enough information about the developer's intention. However if we find the composed pattern "TODO: this is a stupid test", we should carefully analyze this comment and code since it may indicate a TD item. We evaluated this case through an empirical study presented in Chapter 4 in order to implement these compositions, it is necessary that a vocabulary specifies which words can appear together.

Thus, through the relationships proposed by CVM-TD, patterns can be systematically related with each other, resulting in a set of composed patterns. These patterns are represented by Noun Phrases (NP) and semantic expressions. Figure 3.1 shows an overview of the model and illustrates the relationship among the considered dimensions of patterns:

Nouns: These are Software Engineering-related terms. They were extracted from the ontology presented in [82], which is based on various SE domains including programming languages, algorithms, data structures, and design decisions such as design patterns and

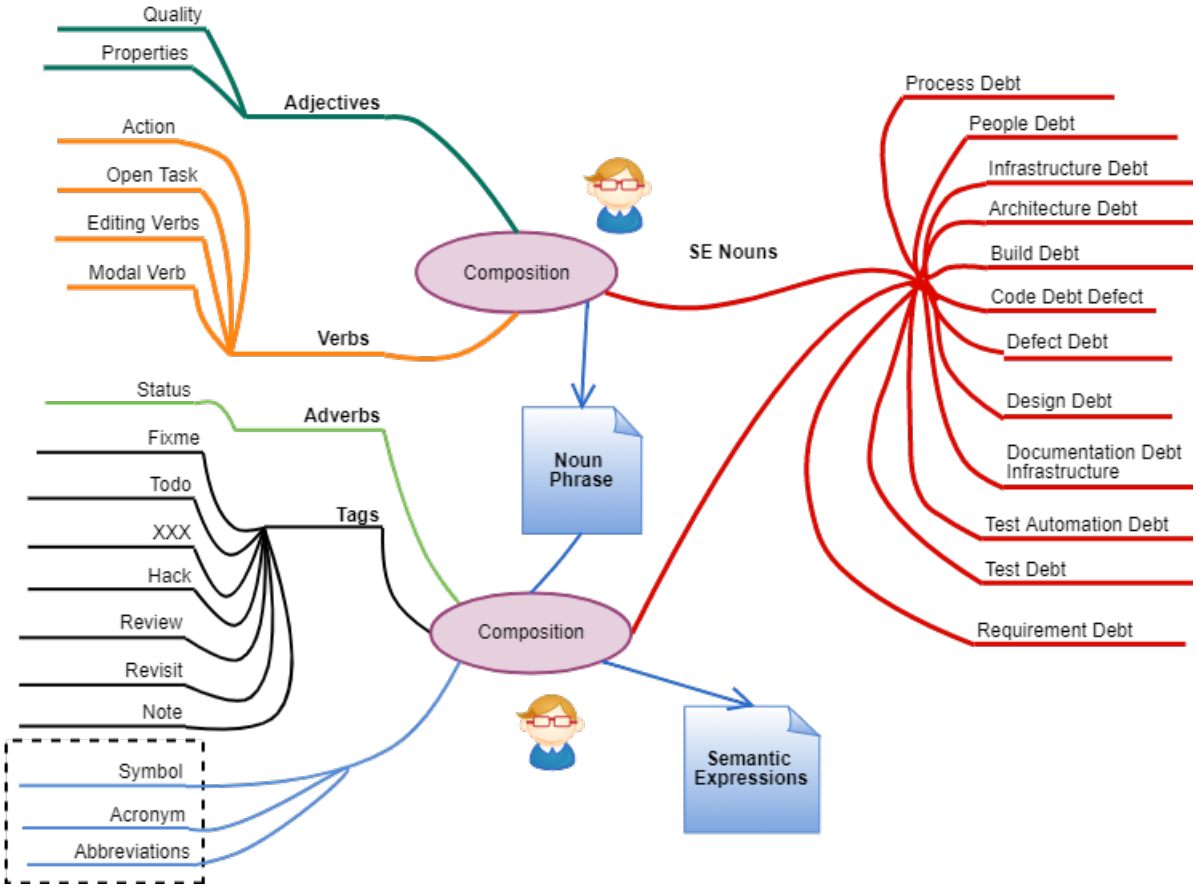


Figure 3.1 Overview of CVM-TD.

software architecture (e.g. code, test, class, method, usage-based techniques, memory and algorithm). Two researchers mapped each Noun to 12 types of TD presented in [13]. The mapping process was performed in two steps. First, each researcher analyzed the list of nouns and chose the related types. Next, the researchers mitigated the divergences in a consensus meeting. Table 3.1 summarizes the relation between types of debt and SE ontological Nouns. A whole list is available at http://goo.gl/5jWY2C.

Adjectives: It can be used to express the quality and properties of a noun and can be used as a phrase modifier [41]. The terms wrong, dangerous, erratic, nuclear, inconsistent, incorrect, obsolete are examples of adjectives (*e.g. the encoding might be wrong. But enough for binary detection*).

Verbs: They are used to denote action in an expression. Action Verbs can denote activities performed or not by developers and refer to software engineering specific activities. They may include terms such as add, insert, duplicate, edit, delete, fix, remove, change, create, test, read, modify review, test, debug, specify, and implement. Modal or auxiliary verbs indicate necessities or possibilities. Some examples are *must, shall, will, should, would, can, could, may and might*. Open task verbs can denote works that need to be completed or that should be done. They may include the following verbs: *to do,*

Table 3.1 Type of TD x SE Nouns.

TD Types	SE Ontological Nouns
Architecture Debt	Coupling, Aspect, Base, Component, Environment, Gui, Function, architecture, Software, System, Query, package,Procedure, Polymorphism, Library, Interface, Hierarchy, Connector, Layer, Link, Message, Module, Tier, ...
Build Debt	Artifact, Aspect, Construction, Deployment, Environment, Polymorphism, Hierarchy, Compilation, Risk, Tool, ...
Code Debt	Algorithm, Code, Component, Deployment, Gui, Query, class, method, file, clone, Metric, Logic, Language, Compilation, Declaration, Variable, Exception, Legibility, Loop, Performance, Rapidity, Release, Version, Speed, ...
Defect Debt	Bug, Deployment, defect, Error, Exception, ...
Design Debt	Component, encapsulation, design, method, metric, code, coupling, Cohesion, Class, clone, Polymorphism, Object, ...
Documentation Debt	Documentation, Diagram, Use case, Template, Model, Manual, Metadata, Remarks, Survey, ...
Infrastructure Debt	HD, Backup, Channel, Data, Database, infrastructure, Structure, Software, System, Screen, Report, Program, Library, Communication, Engineer, Firewall, Hardware, Link, Product, Protocol, Proxy, ...
People Debt	Developer, Programmer, User...
Process Debt	Goal, Business, Analysis, process, Alteration, Allocation, Template, Result, Paradigm, Measurement, Homologation, Cleaning, Communication, Effectiveness, Installation, Interaction, Maintenance, Security, Strategy, ...
Requirement Debt	Scope, Business, Artifact, Conception, Requirements, Specification, ...
Service Debt	Client, Server, Connection, Proxy, Service, ...
Test Debt	Test, Reliability, Precision, Coverage...

need to, should be, must be, can/could/may/might be, would be, should remove, have to and *missing* [16, 41] (e.g. We **need to** refactor the fixtures of tests).

Adverbs: An adverb can be used to denote the status of the work. Some statuses are also important to contextualize software artifacts in the TD context. Examples of adverbs are not, now, nowadays , at the moment, never, instead, only, just, still, incorrectly, enough and still (e.g. *This method incorrectly caches the method structure) [16].*

Tags: They are a specific set of keywords associated with some types of comments. Tags can be found in the main IDEs and customized by developers. We conducted searches in the literature and some open source software (JEdit, ArgoUML, Hadoop, Xerces, and Lucene) looking for tags used by developers. In total, we found 10 different tags: *Fixme, Todo, XXX, Bug, Hack, Remind, Review, Revisit, Note and Remark.* For instance, we found the following comment into jEdit: *"Note: this part of code was not tested as expected"*. If a code was not correctly tested, it might cause future problems in the software maintenance. According to [65], the tag revisit is useful to describe a problem in the short term, but if this piece of code is not revisited quickly, it will remain indefinitely. In the same study, the authors interviewed developers and pointed out that they left TODOs to communicate with team members that something was wrong and that they were aware of the issue.

As can be observed in Figure 3.1, a NP is generated by a semantic composition of words. NPs group words in a context and can improve search accuracy in texts by reducing the difference between comments returned by the vocabulary and those that may indicate a TD item. Generally, a noun is the central element of a NP that determines its syntactical character, and a verb or an adjective modifies this noun [41]. For example, given the patterns "inefficient (Adjective)" and "code (Noun)", the output would be the NP "inefficient code". NPs are patterns that have a high level of SE contextualization.

Semantic expressions are phrases or related words that provide high context information. In order to compose these expressions, we combine adverbs, tags and NPs. For example, given the patterns "TODO (tag)", "should fix (modal verb + verb)" and "inefficient code (NP)", the output would be "TODO should fix inefficient code". The adverb and tag dimensions can also be directly combined with SE nouns in order to generate semantic expressions.

At the end, CVM-TD provides a set of TD patterns (first release of a contextualized vocabulary) that may be used to filter comments that need more attention because they may indicate a TD item. Part of these patterns can be seen in Figure 3.2 (the whole vocabulary and filtered comments is publicly available [27]). The vocabulary is used to filter comments in first experiment (FindTD I) of the family of experiments. It will be discussed in Chapter 4.

3.1 THE EXCOMMENT TOOL

In order to be able to extract, process, and analyze comments from the source code, we developed a tool, named *eXcomment*. The tool was developed based on techniques from text mining, such as data selection, preprocessing, tokenizing the unstructured text, extracting and searching for terms. The process implemented in *eXcomment* has

Figure 3.2 Part of patterns of the vocabulary.

text mining as base. It is important to highlight that *eXcomment* was developed in an incremental form as we can see in Figure 1.2.

Figure 3.3 presents an overview of our tool. The tool was created in two releases and it is divided into three phases: i) preprocessing, ii) search strategy, and iii) classification of TD items and the calculation of final scores of comments. The first release was created in the first study and it searches patterns in comments using a manual strategy. The second release was created from *FindTD III* and it implements the phases of search strategy and classification of the TD items. This release searches the patterns in comments using an automatic strategy.

In the preprocessing phase, comments are extracted from source code and they are processed to filter only those considered useful. This phase is formed by the following steps: data extraction, separation of code and comments, filtering useful comments (those that were intentionally written by developers), joining related comments, gathering the location of comments, and POS process. In the search strategies phase, we used some strategies to find patterns of the vocabulary in the comments. This phase is formed by the steps: removal of stop words, usage of a search engine, applying search heuristics, and identifying new patterns. In the last phase, classification of TD items, the tool calculates the final scores for each processed comment and classifies the types of the TD items identified in the previous phase. This phase is formed by the steps: the calculation of scores for each comment, the calculation of the score of the heuristics, the final scores calculation, and the classification of the TD types. In the following, we present each step in detail. The tool is available at [83].

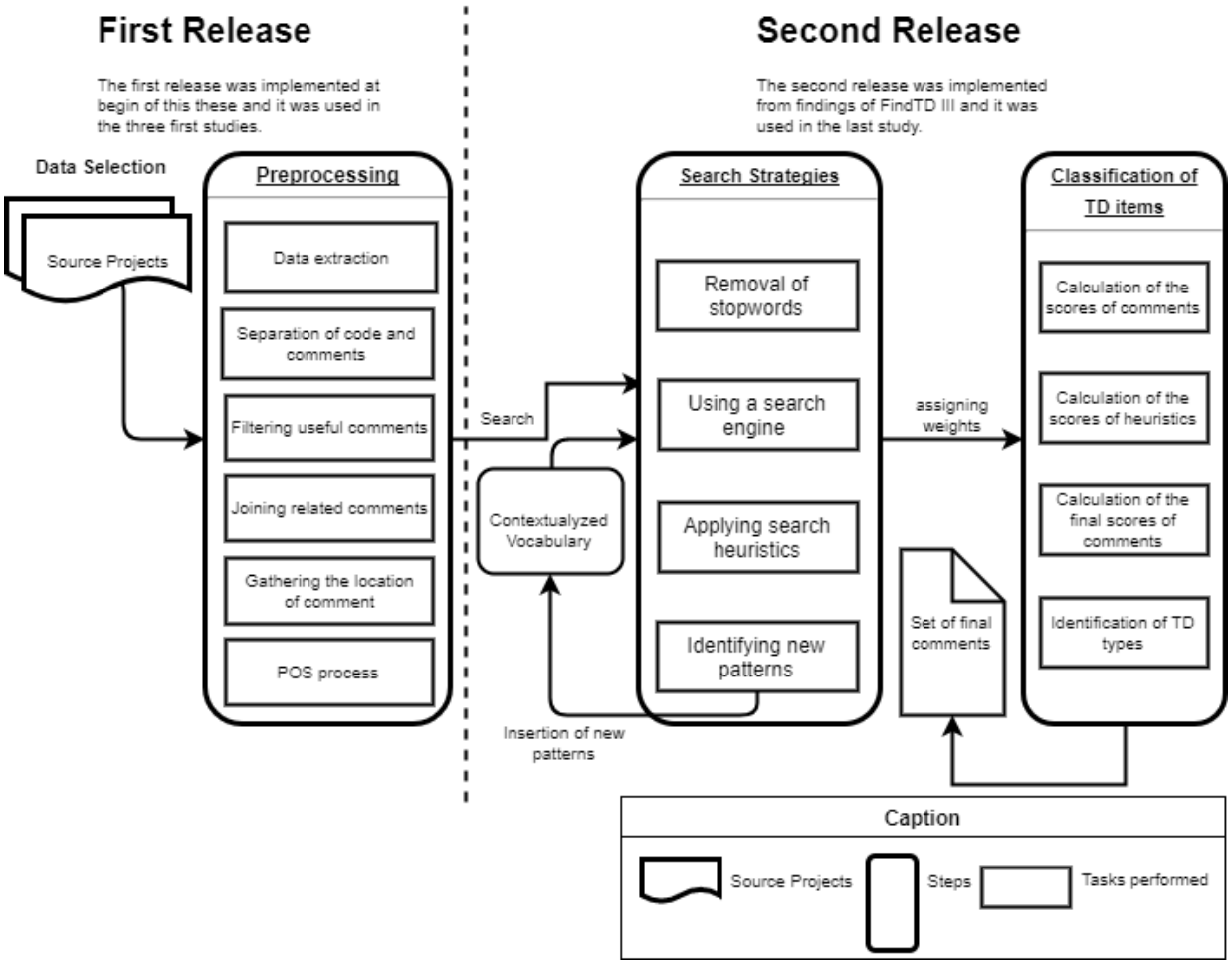


Figure 3.3 Overall overview of our tool

3.1.1 Project data Selection and Extraction (Preprocessing)

eXcomment extracts comments from the source code files of a software. It receives the source code as an input and it parses all files, separating code from comments. The code analyzer parses the source code using the *srcML* toolkit[1].The srcML format is an XML representation for source code, where the markup tags identify elements of the code and comment for the language. The current parsing technologies supports C/C++, C#, and Java. Comments are processed and stored in a relational database system.

Developers use code comments for different purposes in a project, such as giving context, documentation, expressing thoughts, opinions, and disabling the source code from the software. This informal environment allows us to recover insights and confessions of developers about what can be considered as a TD item [2].

As we highlighted in Section 2.4, we are interested only in inline comments and task comments types, which have been intentionally written by developers. *eXcomment* does

[1]http://www.srcml.org/

not consider license and auto-generated comments, copyright or license comments, and also commented code (see some examples in Table 3.2). These types of comments are a majority part of comments in a program and they do not describe a situation of TD [76]. Next, we present how *eXcomment* implements the filtering to remove these types of comments.

License comments. In general, they provide licenses that set the guidelines for use of the code and the things that are copyrighted or have their usage allowed and license comments are commonly added before the declaration of the class. This type of comment does not describe a context of TD. Our tool removes them through a set of expressions used to write a license comment, such as "*This program is distributed*", "*GNU General Public License*", and "*Free Software Foundation*". If more than one of these expressions is found in a comment, we consider it as a license comment. We analyzed several software projects to identify the main expressions describing license comments. The whole list of license expressions is available at [83].

Table 3.2 Discarded Comments.

Project	Discarded comment type	Example of removed comments
jEdit	License	* @author Copyright (c) 1997, 1998 by Microstar Software
jEdit	Commented code	//,if (verbosity >= 4) fprintf (stderr, "sort initialise ...");
Lucene	License	/*Licensed to the Apache Software Foundation (ASF) under one or more. . . */
Lucene	Auto-generated	// TODO Auto-generated method stub

Commented code. They are codes with comment syntax that become a commented code. This may be done to exclude certain parts of the code from the final program due to different reasons. One of them is that it can be used to find the source of an error (debugging). Other is that, the code is not currently being used. Commented source code does not describe a situation of TD. The *eXcomment* removes commented code searching for comment syntaxes combined with the source code.

Auto-generated comments. This type of comment refers to a portion of the comments that are included by an editor program into the code. We removed them using a heuristic to identify the "auto-generated" expressions inserted by IDEs.

Furthermore, some comments are written using multiple line comments. *eXcomment* joins sequential comment lines that are related, keeping the semantic and the context of the comment. An example is in the following:

```
Comment example:
// Todo:  Once we have fixed all subclasses the tittle will
// always be localized so this localization can be removed
```

If the two comments above are analyzed separately, the meaning of comments can be changed so that it can be hard for humans and automated tools to interpret them. To

avoid this issue, *eXcomment* identifies this situation and joins the comment lines, storing only one comment.

To provide detailed information about code comments, and location of any piece of code related to comments, the tool uses abstract syntax tree in order to save information on the method, class, and file where the comment was extracted. This information is useful for knowing what methods and classes present more comments describing TD items.

To validate the filter of comments, we analyze comments from ArgoUML. The strategies presented above reduced the comments to be analyzed in 51.96%, removing the comments that are not useful for identification of TD items. To ensure that the removed comments were really not useful, we manually analyzed a randomly selected set of comments that were classified as license comments, commented code and auto-generated comments. We analyzed 300 comments that were discarded by the filter. We found that 98% of the comments were removed correctly.

In order to quantify the occurrences for each dimension of our vocabulary, *eXcomment* also counts each instance of adjectives, adverbs, nouns, verbs and tags. For that, each comment is analyzed looking for code tags and apply POS process, which tags automatically each word in the comment. For example, given the comment "*TODO: this is a stupid test*", the output is "TODO/*code tag*: this/*pronoun* is/*verb* a/*article* stupid/*adjective* test/*noun*". We used the Stanford Log-linear Part-Of-Speech[2].

3.1.2 Search Strategies

Once the comments are extracted and tagged, we can analyze POS tags and select the comments by using patterns that belong to our vocabulary, creating from CVM-TD model. The vocabulary represents previous knowledge and allows us to recognize parts of the text that are significant to identify TD items through code comments analysis.

A comment is returned when it has at least one pattern of the vocabulary. To do that, we integrate the vocabulary to *eXcomment* in order to automate the TD identification process through code comments mining.

To implement the search, the tool uses some search strategies, such as spellchecking, and tokenizes from Lucene API[3]. We chose it because of its search engine and its API that is able to index a large number of comments. *eXcomment* implements the algorithms *fuzzy search, wildcard search, and proximity search* to find patterns in the comments even when they have not been written exactly equal to the patterns of vocabulary. This strategy allows, for example, *eXcomment* to identify the pattern "*unnecessary*" in a comment that has the term "*unecesary*". Another example is "*Inefficient Code*" that will be identified in the comment that has the expression "*Inefficient Codes*" (please note that the term "*code*" is written in plural).

Another strategy is the removal of stopwords during the search of patterns. For example, in the comment "*TODO: We need to add*", *eXcomment* identifies the pattern "*Todo: need to add*" because the stopword "*we*" was removed from the comment in

[2]Part-Of-Speech: http://nlp.stanford.edu/software/tagger.shtml
[3]Lucene API: http://lucene.apache.org

the search. In another example, the pattern "*Note: code duplicated*" was found in the comment "*Note: the code is duplicated*" because the terms "*the*" and "*is*" were ignored by the *stopwords* filter.

Our vocabulary is composed of combinations of different word classes, such as Tags and Verbs. For example, the pattern "*Todo: Fix*" is composed by Tag "*Todo*" and the Action Verb "*Fix*". In *FindTD III*, we analyzed the combinations composing each pattern. Besides the identification of comments through using the contextualized vocabulary, we also develop and apply selecting heuristics to identify patterns that do not belong to our vocabulary, but can describe a situation of TD.

With that in mind, we created five heuristics that implement the main combinations of vocabulary terms (*"Tag + Action Verb", "Modal Verb + Action Verb", "Open Task Verb + Action Verb", "Adjective + Noun", and "Noun + Is/Are + Adjective"*), resulting in strategies to select comments through the identification of patterns that are not cataloged into our vocabulary, but have the same semantic structure of combinations of patterns that belong to vocabulary. For example, in the vocabulary there are the patterns "*should be*" (Open Task Verb) and "*remove*" (Action Verb), when we apply the heuristic that combines "*Open Task Verbs + Action Verbs*", we can identify the pattern "*should be removed*" even if it does not exist in the vocabulary.

The new patterns identified in this step are automatically inserted into the vocabulary, increasing its power of contextualization. In addition, with more patterns identified by *eXcomment*, our vocabulary is scalable after use. For example, the pattern "*TODO: remove*" does not exist in our vocabulary, but it was found because of the heuristic "*Tag + Action Verb*" so that it was inserted into the vocabulary to be used in next comment mining. We identified 280 new patterns (the whole list is available at [30]) after applying this strategy in *FindTD IV*.

3.1.3 Classification of TD Items and the Calculation of Final Scores of Comments

We propose a process of associating numeric values to each pattern previously identified in comments. This process is known as assigning weights [41]. Each pattern receives an individual score (weight) that indicates its importance in the vocabulary to identify a TD and can be used to calculate the final score of each comment, indicating how much a comment can point out to a TD item. The individual scores were found in *FindTD III*. The participants scored the level of importance for each pattern belonged to our vocabulary. Each pattern was analyzed by at least two participants, ensuring different opinions about it. In more formal writing, each comment is represented as an array, which is composed of elements organized as a tuple of values: $Cj = s1,.., sj$, where Cj represents a comment and sj represents a score associated with each indexed pattern found in each comment.

The final score is calculated through the sum of the scores found in the comment (Si) and the scores of the heuristics (H) found in the comment: $S = (\sum_{i=1}^{j} Si) + H$. An example of the calculation of final score is: in the comment "*// TODO: Redo it (curBody should be as long as curLanguage + button)*", the search strategies found the patterns "*TODO*" with individual scores 2.0 and "*should be*" with individual scores "2.5".

In addition, it found the heuristic "*TAG + Action Verb*" (*TODO: Redo*), which has additional scores 1.2, so that the final score is S = (2.0 + 2.5) + 1.2 = 5.7. The final scores of comments can be used to sort comments on a decreasing scale, showing the comments with higher scores at the top of the list.

The score of a heuristic is the difference between the median of all scores associated with a combination and the score of the patterns that are found alone. For example, the median of the combination "*Tag + Action verb*" is 3.0, whereas the score of the patterns classified as *Tag* is 1.8. So, the score of this heuristic is *3.0 – 1.8 = 1.2*. We do not ponder the scores of the "Action Verb" patterns because they are not found alone.

Table 3.3 shows the heuristics we implemented in *eXcomment*, a comment example identified by each of them in ArgoUML, and its scores. The heuristics' scores are used to calculate the final scores of each comment.

Table 3.3 Heuristics implemented in eXcoment.

Heuristics	Comment example	Heuristic scores
TAG + Action Verb	// TODO: remove when code below in characters() is removed	1.20
Modal Verb + Action Verb	// hack for to do items only, should check isLeaf(node), but that // includes empty folders. Really I need alwaysLeaf(node).	1.20
Open Task Verb + Action Verb	// need to update the selection state.	0.60
Adjective + Noun	//Break up one complex method into a few simple ones and // give the diagrams more knowledge of themselves	1.10
Noun + IS/ARE + Adjetive	// TODO: This method is obsolete. Use getInputMap etc as below	1.40

Relation between patterns, TD themes and types of TD. We regarded the TD indicators proposed by [13] as guideline. TD indicators allow the discovery of the type(s) of TD items when analyzing the different artifacts created during the development of a software project. The tool analyzes the patterns found in each comment in order to identify TD indicators. To do that, we used coding technique to classify the patterns into categories related to different situations of TD, which we defined as TD themes. There is at least one pattern associated with each theme (possibly more), and a pattern can be associated with more than one TD theme and vice versa.

Next, we map TD themes and TD indicators, considering information provided by the definitions and scope of each TD type and its indicators. It is also important to mention that the indicators were organized by the TD types that they are associated with. Thus, when we associate a patter with a TD theme and a TD theme with a TD indicator, we are also associating the pattern and the TD theme with TD types. We applied these relationships aiming to identify comments related to specific types of TD, for example, the theme "dependencies" was associated with the indicator "structural dependencies"

and this indicator is associated with architecture and build debt, consequently "dependency" is also associated with architecture and build debt. *eXcomment* implements these relationships and links the patterns and comments to one or more type(s) of TD. If the *eXcomment* is not able to associate the identified patterns to a TD indicator, a comment is linked to the description "none" type of TD. We explored and evaluated this approach using a family of experimental studies.

Next four chapters present an overview the family of experiments that we carried out in order to understand how code comment analysis can be explored to identify and classify TD items (*FindTD I, FindTD II, FindTD III, FindTD IV*). The primary purpose of the experiments is investigating, evaluating and incrementally evolving our approach. They detail each experimental study setup and presents the results, discussions, and threats to validity.

Chapter 4

This chapter overviews the first experiment of the family of experiments that we carried out as an exploratory study on two large open sources projects with the goal of characterizing the feasibility of the proposed model to support the detection of TD through code comments analysis. The results showed that the proposed model provided a vocabulary that can be used to identify TD items.

FINDTD I

4.1 GOAL OF STUDIES AND RESEARCH QUESTIONS (RQ)

We performed an exploratory study to **analyze** CVM-TD, **with the purpose of** characterizing, **with respect to** the feasibility of the proposed model to support comments analysis for identifying TD, **in the context of** software development projects, **from the point of view of** software engineering master students with professional experience. The experiment has been performed as an exploratory study aiming to characterize the feasibility of the proposed model to support the detection of TD through code comments analysis.

4.2 PROCEDURE

First, we selected projects and participants for the exploratory study. We chose the jEdit[1] and Lucene[2] as project to be studied and four master students to analyze the comments. As the second step, we mined the source code from the selected projects to extract their comments. Each word of each comment was categorized according to code tags and word classes. Once the comments were extracted and tagged, we analyzed POS tags and filtered the comments by using patterns that belong to the first release of the vocabulary proposed by CVM-TD and presented in Chapter 3. Third, for each CVM-TD dimension, we counted how many adjectives, adverbs, verbs, nouns, and tags were identified. We used the *eXcomment* tool to carry out the steps 2 to 3 and provide a list of candidate comments. Candidate comments are those that can indicate a situation of TD. We also used these patterns in next experiments.

[1]jEdit: http://www.jedit.org/
[2]Lucene: https://lucene.apache.org/

In the last step, with the final list of comments ready to be analyzed, we asked four software engineering master students with professional experience to analyze each of them in order to indicate those that may indicate a TD item. Figure 4.1 illustrates this process.

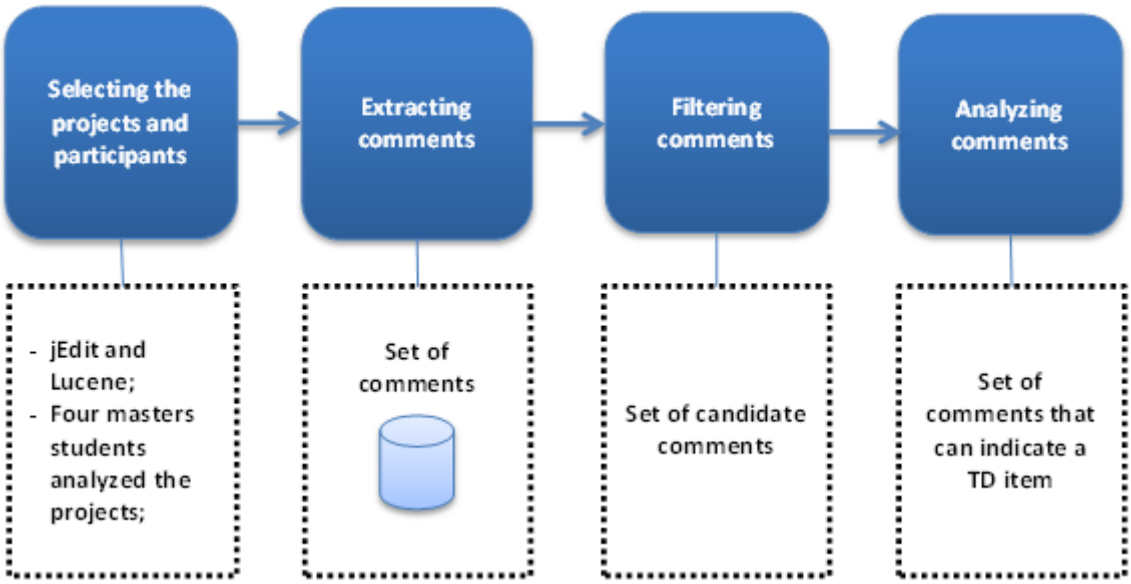


Figure 4.1 Process of FindTD I.

Specifically, we analyzed which model dimensions and patterns included in the vocabulary appear frequently in the two software projects and whether the filtered comments are significant to identify a TD item.

4.3 SELECTED PROJECTS AND PARTICIPANTS CHARACTERIZATION

We gathered and analyzed comments from two large and well-known open source software (OSS) – jEdit 4.3.2 and Apache Lucene 4.3.0. Both projects are written in Java with 531 and 3,037 Java files, respectively. For choosing these projects, we considered the following criteria: be Long-lived (more than 10 years), have a satisfactory number of comments (more than 2,000 useful comments). Table 4.1 summarizes their metadata.

Table 4.1 Software Metadata.

Software	Age (years)	# of code lines	# of comment lines	# of useful comments
JEdit 4.3.2	15	281,830	68,145	2,006
Lucene 4.3.0	15	437,205	156,320	4,259

Two participants analyzed each project. All participants have more than six years on software development experience (three of them as developer and software engineer,

and one of them as requirements analyst). Besides, three of them have experience on refactoring activities.

4.4 ANALYSIS OF FINDTD I

Table 4.2 summarizes POS tagging and code tags that we have found in jEdit and Lucene. The columns '# of comments' and '# of patterns' indicate the total of comments that have a specific POS category and the occurrences of a POS item, respectively. '% of comments' and '% of patterns' indicate the proportion of each POS item. To calculate the ratio, we consider the total number of comments that each project has. Thus, if jEdit has 2,006 comments and Lucene has 4,259, and 1% of both projects' comments contain at least one Adjective, then they have 20 and 42 comments with Adjectives, approximately.

Table 4.2 Summary of the Parts of Speech and Code Tags.

jEdit – Total of comments: 2,006				
POS	**# of comments**	**# of patterns**	**% of comments**	**% of patterns**
Adjectives	1,262	2,427	63.64	7.33
Verbs	1,735	5,299	87.49	16.01
Adverbs	879	1,468	44.33	4.44
Nouns	1,829	9,238	92.23	27.91
Tags	199	230	10.04	0.69
Lucene – Total of comments: 4,259				
POS	# of comments	# of patterns	% of comments	% of patterns
Adjectives	3,294	6,363	77.51	15.03
Verbs	2,130	5,194	50.12	12.27
Adverbs	898	1,095	21.13	2.59
Nouns	3,829	22,777	90.09	53.81
Tags	241	257	5,67	0,61

Adjectives: we found 1,262 (63.64%) comments that include Adjectives in jEdit and 3,294 (77.51%) in Lucene. There are 2,427 (7.33%) and 6,363 (15.03%) Adjective occurrences in jEdit and Lucene, respectively. These values show that a large amount of comments include Adjectives expressing properties in its content. For instance, "*there must be a better way of fixing this (jEdit)*" and "*query does not match doc 1 because "gradient*" is in wrong place there".

Adverbs: Adverbs are used significantly less than other word classes analyzed in this study. In jEdit, we found 879 comments that have any Adverb, and 898 in Lucene. Examples are: *"Should never come here (jEdit)*" and "*Not enough space for match 2 : remove it (Lucene)*".

Verbs: we found 5,299 (16.01%) Verb instances in jEdit and 5,194 (12.27%) in Lucene. In order to discuss each Verb category present in our model, Table 4.3 provides a summary of the most common Verbs by category. From it we can see that:

(i) Edit and action Verbs appear more frequently with 11.20% (jEdit) and 17.50% (Lucene) of Verbs. In this category, Verbs add, read, and fix are the most common ones.

Some examples of such comments are: "*Add new constants for later versions **here** to respect order! (Lucene)", "How many blocks we've read since last seek... (Lucene)" and "need to "fix" the caret position so that we get the right rule*".

(ii) Modal Verbs represent 9.10% in jEdit and 8.47% in Lucene of all Verbs in our comments base. The most used are can, should and will in both OSS. Some examples are: *"this can be a relativly heavy operation depending on the delegate postings format and their underlying directory (Lucene)", "??? this should only be called with fully loaded action set (jEdit)", "it should not be possible to have a sentence with 0 words, check just in case (Lucene)"* and *"in the future we will want something better, eg. having an 'evt' object passed to EditAction.invoke() (jEdit)".*

(iii) Finally, Open Task Verbs represent 3.68% (jEdit) and 2.75% (Lucene) of all Verb instances. Some instances extracted from OSS are: *"Note: incorrect here - we are using the hack in bsh class manager that allows lookup by base name. We need to eliminate this limitation... (jEdit)", "could be more efficient (Lucene)", "TODO: this should be private to the codec, not settable here (Lucene)", "his must be fixed during plugin development (jEdit)"* and *"have to be careful, retrieveTerms returns all words but that's probably not useful to our caller...(Lucene)".* The most common are *need to, can/could/may/might be, should be, must be and have to.*

Table 4.3 Most Common Verbs by Category.

Edit and action Verbs	% of Instances jEdit 11.12	% of Instances Lucene 17.50	Modal Verbs	% of Instances jEdit 9.10	% of Instances Lucene 8.47	Open Task Verbs	% of Instances jEdit 3.68	% of Instances lucene 2.75
add	1.42	2.62	can	31.12	37.27	need to	28.72	24.48
read	1.83	4.43	should	20.54	22.27	can/could/ may/might be	25.13	18.88
fix	1.26	4.43	will	18.26	15.68	should be	12.82	16.78
change	1.30	0.83	must	9.34	12.73	must be	11.79	16.78
test	0.59	4.12	could	7.68	8.64	have to	10.26	4.20
create	1.06	0.90	may	6.22	7.73	to do	7.69	5.59
remove	0.85	1.02	would	4.15	7.50	would be	1.54	9.79
delete	0.28	1.31	might	6.22	2.27	missing	0.51	3.50
debug	0.74	0.46	shall	–	0.91	would have	1.54	–

These three types of Verbs represent 63.81% (jEdit) and 56.56% (Lucene) of all comments that have any Verb in its content.

Nouns: Noun is the most used word class in comments written by developers in both OSS. We identified 1,829 (92.23%) comments that include any Noun in jEdit and 3,829 (90.09%) in Lucene. Table 4.4 shows the Noun frequencies by type of debt.

The two types of debt that include the most frequently cited Nouns in both projects are code and design debt. Examples are: *"error messages for the codes above (Lucene)"*

and *"the vfs browser has what you might call a design flaw, it doesn't update properly..."* (jEdit). This indicates that these two types are concerns to the development team of these projects and may be explored to try to detect code and design debts. In addition, Nouns related to architecture and test debt (e.g. *"Note this is really a stupid test just to see if things arent horribly slow"*) are also usually referenced in comments.

Table 4.4 Frequency of Nouns by Types of TD.

Contextualized Noun	# of Noun occurrences	
	jEdit	Lucene
Code debt	604	490
Design debt	441	134
Architecture debt	80	116
Test debt	23	154
Defect debt	66	63
Infrastructure debt	132	36
Process debt	30	22
Documentation debt	5	9
Build debt	7	2
Requirement debt	9	2
Service debt	18	1
People debt	54	0

Unexpectedly, we also have a high number of Nouns related to infrastructure and people debt. Infrastructure debt refers to infrastructure issues that, if present in the software organization, can delay or hinder some development activities (some examples of this kind of debt are delaying an upgrade or infrastructure fix) [6]. People debt refers to people issues that, if present in the software organization, can delay or hinder some development activities. An example of this kind of debt is the expertise concentrated in too few people, as an effect of delayed training and/or hiring [6]. It is not usual to have comments about these kinds of debt in source code. This result indicates that the mapping between SE Nouns and TD types needs to be better calibrated.

Tags: as it can be seen from Table 4.5, the Tag "Todo" had the highest frequency of use in the Lucene's comments and the Tag *Note* in the jEdit's comments. On the other side, *Remind, Review*, and *Revisit* are the less used. According to [65], the meaning of these tags is based on informal conventions they have individually assumed or informally agreed on within their team. In general, they are used to describe things that must be completed or issues that may require work in the future. In this sense, they may point to a TD. For example, "*Note: This class is messy.... These should be rewritten to avoid having to catch the exceptions. Method lookups are now cached at a high level so they are less important, however the logic is messy (jEdit)".*

We identified 109 comments selected by the participants from jEdit and 152 from Lucene. It indicates that 12.87% (jEdit) and 17.31% (Lucene) of the filtered comments may be a TD indicator. Table 4.6 shows some examples. In addition, the participants reported that there are some comments that do not clearly indicate a TD item but need to be investigated. The whole set of selected comments is available at http://goo.gl/HBc5nt

Table 4.5 Code Tag Used in the Analyzed Software Projects.

Tags	# of Tag occurrences	
	jEdit	Lucene
Fixme	3	3
Todo	11	158
Bug	62	51
Note	102	42
XXX	23	0
Hack	23	2
Remind	1	0
Review	5	0
Revisit	0	1
Remark	0	0

4.5 SUMMARY OF FINDINGS AND INSIGHTS

In summary, we notice that the dimensions considered by the model (e.g. Adjective, Nouns, etc) are used by developers when writing comments and CVM-TD provides a vocabulary that makes it possible to extract a list of useful comments to support TD identification. The mapping between TD types and SE Nouns is an important contribution because these Nouns can be used to classify the TD item.

However, this version of the mapping between SE Nouns and TD types needs to be better calibrated in order to reduce the difference between comments returned by the vocabulary and those that may indicate a TD item. In general, the results provide some evidence and motivation to continue exploring code comments in the context of TD identification process.

Although the results show evidence that the model and vocabulary can filter TD comments, it is necessary to evaluate accuracy when classifying candidate comments and factors that influence the analysis of the comments to support the identification of TD. To do that we performed a controlled experiment (*FindTD II*) to further evaluate CVM-TD and the vocabulary with other data sources.

Table 4.6 Selected Comments

jEdit	
Comments	**Class.Method**
// FIXME: Here the method reset() can fail if the // previous detector read more than buffer size of // markedStream.	AutoDetection.getDetectedEncoding
// REVIEW // This is horribly inefficient, but it ensures that we // properly skip over bytes via the TarBuffer... //	TarInputStream.skip
/* Note: this implementation is temporary....	BshClassManager.definingClass
/* Note: bshmethod needs to re-evaluate arg types here This is broken. */	BshMethod.getParameterTypes
// TODO: // ... we need to refactor out this common functionality and make sure ...	BSHPrimarySuffix.doName
/* Catch the mismatch and continue to try the next Note: this is innefficient, shoul have an isAssignableFrom() that doesn't throw */	BSHTryStatement.eval
Lucene	
Comment	**Class - Method**
// a problem here is from clear() or nextSetBit	TestFixedBitSet.doRandomSets
// large query so that search will be longer	TestTimeLimitingCollector.setUp
// TODO: maybe take List here?	MultiPassIndexSplitter.split
// TODO doing this each time is not necessary maybe	DocumentsWriterDeleteQueue.add
// well we could, but this is stupid	ByteBufferIndexInput.Slice
// could be more efficient	CharArrayMa.put

4.6 THREATS TO VALIDITY

We followed the checklist provided by [84] to discuss the relevant threats to this controlled experiment.

4.6.1 Construct Validity.

The vocabulary created by CVM-TD may not be fully representative to determine comments that contain a TD indicator. To help reduce this threat, our model was presented and argued with TD specialists before be used in the exploratory study. Another risk involves the definition of the proposed vocabulary. It is possible that the set of patterns and combinations used by our model and vocabulary are simply too many to be studied. An alternative would be to limit the studies to a specific contexts and software domains. Finally, to reduce social threats due to evaluation apprehension, participants were not evaluated.

4.6.2 Internal Validity.

The first internal threat we have to consider is participant selection, since we have chosen all participants through a convenience sample. We minimized this threat choosing the participants randomly to perform the analysis of comments. Another threat is that participants might be affected negatively by boredom and tiredness. In order to mitigate this threat, we performed a pilot study to calibrate the time and task to be performed by participants. A further validity threat is the instrumentation, which is the effect caused by artifacts used for the experiment. Each two participants had a specific set of comments from a OSS project, but all participants used the same data collection form format.

4.6.3 External Validity.

Next limitation of the studies is that even all data was gathered from large OSS projects, the findings may not be generalized to other software, including industry projects. At this point, we cannot state whether there could be any difference in the used patterns of the vocabulary if we repeat our studies analyzing industrial projects. In spite of the fact that used software are mature and large projects, and our results seem to be quite consistent with the goal of characterizing the model feasibility to support TD identification, we still need to perform further investigation. A further threat is the usage of software that may not be representative for industrial practice. We used software adopted in the practice of software development as an experimental object in order to mitigate the threat.

4.6.4 Conclusion Validity.

Conclusion validity threats are mainly due to avoid the violation of assumptions. In this first exploratory study, we limited our data analysis at considering simple statistic methods. Another point is the size of sample. The findings considered the data analysis of four participants in the experiment in two OSS projects. Our sample cannot be considered as appropriate for a first exploratory investigation but the next studies intend to explore

other projects and participants in order to validate of the conclusions drawn of this study.

Chapter 5

This chapter presents the second study of the family of experiments that we carried out as a controlled experiment. We investigate the use of CVM-TD with the purpose of characterizing factors that affect the accuracy of the identification of TD, and the most chosen patterns by participants as decisive to indicate TD items. We also investigated if the contextualized vocabulary provided by CVM-TD points to candidate comments that are considered indicators of technical debt by participants. The results indicated that CVM-TD offered promising results considering the accuracy values. We identified a list of the 20 most chosen patterns by participants as decisive to report TD items. The results motivate us to continue to explore code comments in the context of TD identification process in order to improve CVM-TD.

FINDTD II

5.1 GOAL OF STUDIES AND RESEARCH QUESTIONS (RQ)

This study aims to **analyze** the use of CVM-TD **with the purpose of** characterizing, **with respect to** overall accuracy and factors affecting the identification of TD through code comment analysis **from the point of view of** the researcher **in the context of** software engineering master students with professional experience analyzing candidate code comments of large software projects. More specifically, the initial outcomes from *FindTD I* motivated us to further evaluate CVM-TD with other projects. Therefore, in this experiment we extend the *FindTD I* with an additional study to analyze the use of CVM-TD and the contextualized vocabulary with the purpose of characterizing its overall accuracy when classifying candidate comments and factors that influence the analysis of the comments to support the identification of TD in terms of accuracy.

We address this research goal by conducting a controlled experiment. We analyzed the factors against the accuracy by observing the agreement between each participant and an oracle elaborated by the researchers. We compared the accuracy values for the different factors using statistical tests. We analyzed the agreement among the participants. We also identified a set of new patterns from comments, which we intend to evaluate in next experiment. These aspects are decisive to understand and validate the model and the contextualized vocabulary.

More specifically, we investigated five RQs. The description of these RQs follows:

RQ1*: Do the English reading skills of the participant affect the accuracy when identifying TD through code comment analysis?*

Considering that non-native English speakers are frequently unaware of the most common patterns used to define specific parts of code in English [85], this question aims to investigate whether a different familiarity with the English language could impact the identification of TD through code comment analysis. In order to analyze this variable, we split the participants into levels of "good English reading skills" and "medium/poor English reading skills". This question is important to help us understand the factors that may influence the analysis of comments to identify TD.

RQ2*: Does the experience of the participant affect the accuracy when identifying TD through code comment analysis?*

Experience is an important contextual aspect in the software engineering area [86]. Recent research has studied the impact of experience on software engineering experiments [87]. Some works have found evidence that experience affects the identification of code smells, and that some code smells are better detected by experienced participants rather than by automatic means [88]. Considering this context, this question aims to discuss the impact of the participants' experience on the identification of TD through code comment analysis. In order to analyze the variable, we classified the participants into three levels considering their experience with software development: i) high experience, ii) medium experience, and iii) low experience. This question is also important to help us to understand the factors that may influence the analysis of comments to identify TD.

RQ3*: Do participants agree with each other on the choice of comments filtered by CVM-TD that may indicate a TD item?*

With this question, we intend to investigate the contribution of CVM-TD in the TD identification process and how many and what comments achieve higher levels of agreement. In other words, what comments point out to a TD item. This will also allow us to analyze the agreement among the participants about the candidate comments that indicate a TD item. our assumption is that a high agreement on the choice of comments filtered by CVM-TD provides evidence on its relevance as a support tool on the TD identification.

RQ4*: Does CVM-TD help researchers in the selection of candidate comments that point to technical debt items?*

With this question, we intend to investigate if the contextualized vocabulary provided by CVM-TD points to candidate comments that are considered indicators of TD by researchers. This will also allow us to investigate the contribution of CVM-TD to support the TD identification process.

RQ5*: What were the most chosen code comment patterns by participants as decisive to indicate a TD item?*

In this question, we intend to investigate which patterns were considered more decisive to help participants on identifying comments that report a situation of TD. For each comment marked as yes, the participants highlighted the part of the comment that was decisive for their answer. The answer for this question will allow us to classify the most decisive patterns and closing a feedback cycle expanding the vocabulary by inserting new patterns.

Figure 5.1 summarizes the process of this study. First, we selected one project and participants for the controlled experiment. We chose ArgoUML OSS as project to be

studied and 36 participants to analyze the comments. As the second step, we mined the source code from the selected project to extract their comments. Next, comments were filtered by using patterns that belong to the second release of the vocabulary. This step provided a set of candidate comments. In next step, three TD specialists built an oracle of the comments that really may report a TD item. Finally, we asked participants to analyze each candidate comment in order to indicate those that may describe a TD item and the most important patterns.

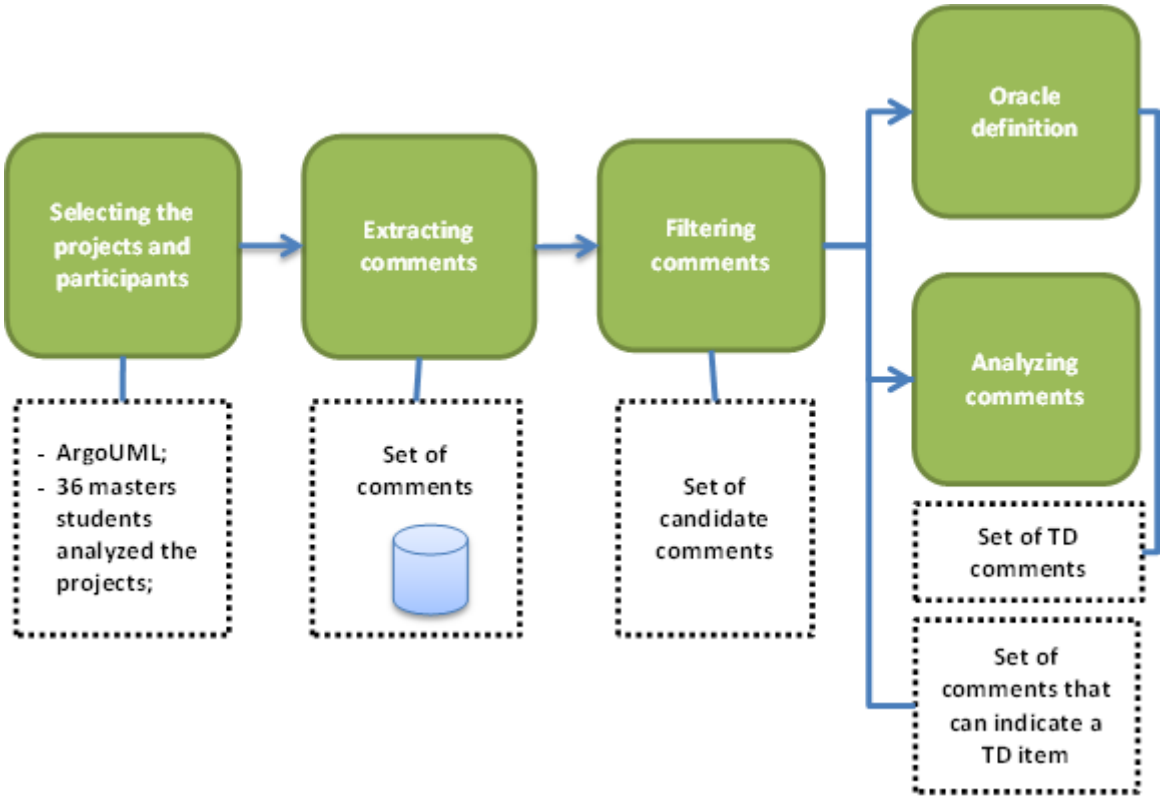


Figure 5.1 Process of FindTD II.

5.2 PARTICIPANTS

The participants were selected using convenience sampling [20]. Our sample consists of 21 software engineering master students at the Federal University of Sergipe (Sergipe-Brazil) and 15 software engineering master students at the Salvador University (Bahia-Brazil). In order to classify the profile of the participants and their experience in the software development process, a characterization form was filled by each participant before the experiment. The questions were about professional experiences, English reading skills, and specific technical knowledge such as refactoring and programming languages. The results of the questionnaire showed that participants had a heterogeneous experience level, but all had some type of experience on software projects.

The participants were classified into three experience levels (high, medium and low) regarding the experience variable and the classification proposed by [86], which is presented in Table 5.1. We discarded the category E1 because there were not any undergraduate students as participants in both experiments. We considered low experience for participants related to the categories E2 and E3. The participants related to the category E4 were considered as having medium experience, and, finally, we considered the participants related to category E5 as having high experience.

Table 5.1 Classification of the experiences of participants.

Category	Description	Experience levels
E1	Undergraduate student with less than 3 months of recent industrial experience	–
E2	Graduate student with less than 3 months of recent industrial experience	Low
E3	Academic with less than 3 months of recent industrial experience	Low
E4	Any person with recent industrial experience between 3 months and 2 years	Medium
E5	Any person with recent industrial experience for more than 2 years	High

When considering the English reading skills, the participants were classified into two levels (good and medium/poor). We had 4 participants with poor English reading skills, and 21 participants with medium. Despite these participants have been selected as medium/poor English, they may understand short sentences like code comments in English. Table 5.2 shows the characterization of the participants. The participants were split into three groups. Each group had 12 participants with approximately the same levels of experience. This strategy provides a balanced experimental design. The design involved each group of participants working on a different set of comments (experimental object) and it allowed us to use statistical test to study the effect of the investigated variables. We adopted this plan to avoid an excessive number of comments to be analyzed by each participant.

5.3 INSTRUMENTATION

5.3.1 Forms.

The experimental package is publicly available at [28]. We used slides for the training and four forms to perform both experiments:

- *Consent form*: the participants authorize their participation in the experiment and indicate to know the nature of the procedures which they have to follow.

- *Characterization form*: contains questions to gather information about professional experiences, English reading skills, and specific technical knowledge of participants.
- *Feedback form*: in this form, the participants may write their impression on the experiment. We also asked the participants to classify the training and the level of difficulty in performing the study tasks.
- *Data collection form*: contains a list of source code comments. During the experiment, the participants were asked to indicate, for each comment, if it points to a TD item and to mark the chunks of text that they believe to be important to report a TD context.

Table 5.2 Distribution of the participants of FindTD II.

Group	Partic. by experience level			Partic. by English reading level	
	High	Med	Low	Good	Med/Poor
G1 (12)	4	3	5	1	11
G2 (12)	3	5	4	5	7
G3 (12)	4	5	3	5	7
Total (36)	**11**	**13**	**12**	**11**	**25**

5.3.2 Software Artifact and Candidate Comments.

We gathered and filtered comments from a large and well-known open source software (ArgoUML). The project is written in Java with 2,609 files and 27,298 comments. In choosing this project, we considered the following criteria: being long-lived (more than 10 years), having a satisfactory number of comments (more than 2,000 useful comments).

To be able to extract the candidate comments from the software that may indicate a TD item, we used *eXcomment*. We were only interested in comments that have been intentionally written by developers, as discussed in Section 2.4.

Once the comments were extracted, we filtered the comments by using patterns that belong to the vocabulary. A comment is returned when it has at least one keyword or expression found in the vocabulary. We will call these comments 'candidate comments'. At the end, the tool returned 353 comments, which were listed in the data collection form in the same order in which they are in the code. This is important because comments that are close to each other can have some kind of relationship.

5.4 ANALYSIS PROCEDURE.

We considered three perspectives to analyze accuracy:

Agreement between each participant and the oracle: In order to investigate RQ1 and RQ2, we adopted the accuracy measure, which is the proportion of true results (the comments chosen in agreement between each participant and the oracle) and the total number of cases examined (see Equation 5.1).

$$accuracy = \frac{(numTP + numTN)}{(numTP + numFP + numTN + numFN)} \quad (5.1)$$

TP represents the case where the participant and the oracle agree on a TD comment (comment that points to a TD item). FP represents the case where the participant disagrees with the oracle with respect to the selected TD comment. TN occurs when the participant and the oracle agree on a comment that does not report a TD item. Finally, a FN happens when the participant does not mark a TD comment in disagreement with the oracle.

The definition of the oracle, which represents an important aspect of this analysis process, was performed prior to carrying out the experiment. We relied on the presence of three specialists in TD. Two of the specialists did, in separate, the indication of the comments that could point out to a TD item. After, the third specialist did a consensus process for the set of the chosen comments. All this process took one week.

Agreement among the participants: To analyze RQ3, we adopted the Finn coefficient [89]. The Finn coefficient is used to measure the level of agreement among participants. In order to make the comparison of agreement values, we adopted classification levels, as defined by [90], and recently used by [88]: slight, for values between 0.00 and 0.20; fair (between 0.21 and 0.40); moderate (between 0.41 and 0.60); substantial (between 0.61 and 0.80); and almost perfect (between 0.81 and 1.00) agreement.

TD comments selected by oracle: To analyze RQ4, we investigated the candidate comments that point to TD items selected by the oracle and participants. We also identified and analyzed the false positives (i.e., comments that do not report a TD item).

The most chosen patterns by participants as decisive to indicate a TD item: To answer the RQ5, we analyzed the patterns that were chosen by participants as decisive patterns to identify a TD item through code comments analysis.

5.5 PILOT STUDY

Before the experiment, we carried out a pilot study with a computer science PhD student with professional experience. The pilot took 2 hours and was carried out in a Lab at the Federal University of Bahia (Bahia-Brazil). We performed the training at the first hour, and next the participant performed the experimental task described in the next section for each experiment. The participant analyzed 83 comments and selected 52 as TD comments.

The pilot was used to better understand the procedure of the study. It helped us to evaluate the use of the data collection form, the average time to accomplish the task and, mainly, the number of comments used by each group in the experiment. Thus, the pilot study was useful to calibrate the time and number of comments analyzed.

5.6 OPERATION

The experiment was conducted in a classroom at the Federal University of Sergipe, and at the Salvador University, following the same procedure.

The operation of the experiment was divided into different sessions. A week prior to the experiment, the participants filled the consent and characterization form. The training and experiment itself were performed on the same day. For training purposes, we performed a presentation in the first part of the class. The presentation covered the TD concepts and context, as well as the TD indicators [13] and how to perform a qualitative analysis on the code comments. This training took one hour.

After that, a break was taken. Next, each participant analyzed the set of candidate comments, extracted from ArgoUML, in the same room where the training was provided. They filled the data collection form pointing out the initial and end time of the task. For each candidate comment listed in the form, the participants chose "Yes" or "No", and their level of confidence on their answer. Besides, for each comment marked as yes, they should highlight the piece of text that was decisive for giving this answer.

The participants were asked to not discuss their answers with others. When they finished, they filled the feedback questionnaire. A total of three hours were planned for the experiment training and execution, but the participants did not use all of the available time.

5.6.1 Deviations from the Plan.

We did not include the data points from participants who did not complete all the experimental sessions in our analysis since we needed all the information (characterization, data collection, and feedback). Thus, we eliminated 4 participants.

Table 5.3 presents the final distribution of the participants. The value in parentheses indicates the final number of participants in each group. In each of the groups G1 and G3, one participant was excluded because she did not fill the value of confidence. In group G2, one participant was excluded because she did not analyze all comments and another was excluded because did not mark the text in the TD comments.

Table 5.3 Final distribution of the participants among groups in FindTD II.

Group	Participants by experience level			Part. by English reading level	
	High	Med	Low	Good	Med/Poor
G1 (11)	4	3	4	1	10
G2 (10)	2	5	3	5	5
G3 (11)	3	5	3	5	6
Total (32)	**9**	**13**	**10**	**11**	**21**

5.7 ANALYSIS OF FINDTD II

In this section, we present the results for each RQ in *FindTD II*.

5.7.1 The impact of the English reading level skills on the TD identification process (RQ1).

In order to investigate the impact of the English reading level skills on the TD identification process, we calculated the accuracy values for each participant with respect to the oracle. Figure 5.2 shows a box plot illustrating the accuracy distribution. It is possible to note that the participants with good English reading skills had the lowest dispersion. It indicates that they are more consistent in the identification of TD comments than the participants with medium/poor English reading skills. Moreover, the accuracy values of the participants with good reading skills are higher than the values of the participants with medium/poor reading skills. However, the median accuracy of the participants with medium/poor reading skills is 0.65. This means that the participants with this profile were able to identify comments that were pointed out as an indicator of a TD item by the oracle.

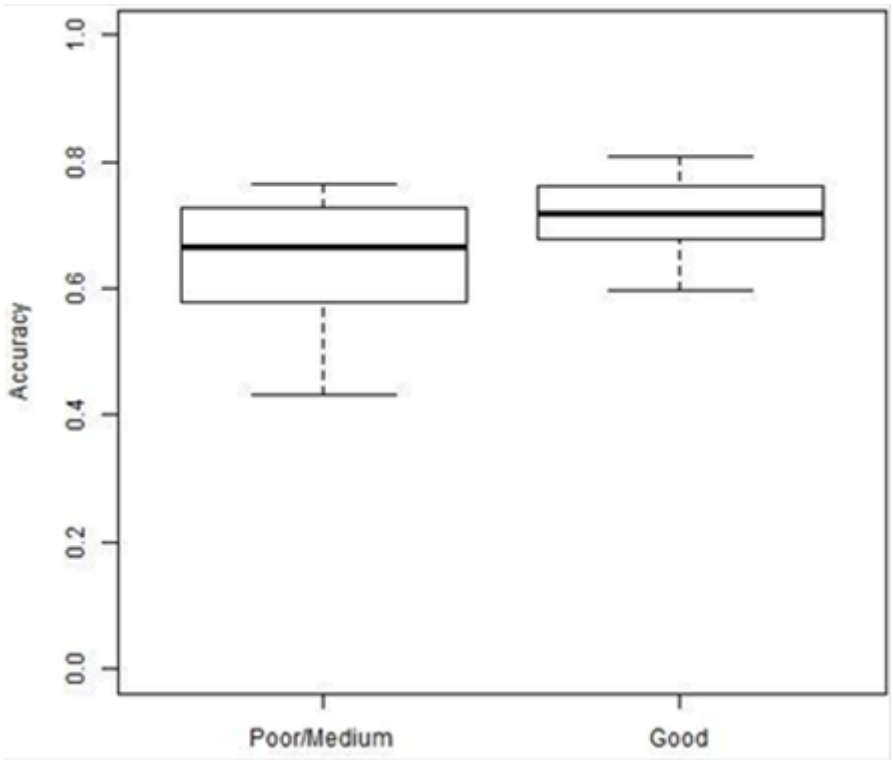


Figure 5.2 Accuracy value by English reading skills.

We also performed a hypothesis test to reinforce the analysis of this variable. To do this, we defined the following null and alternative hypotheses:

H0: The English reading skills of the participant do not affect the accuracy with respect to the agreement with the oracle.

H1: The English reading skills of the participant affect the accuracy with respect to the agreement with the oracle.

We ran a normality test, Shapiro-Wilk, and identified that the distribution was nor-

mally distributed. After that, we ran the t-test, a parametric test, to evaluate our hypotheses. We used a typical confidence level of 95% ($\alpha = 0.05$). As shown in Table 5.4 , the p-value calculated (p=0.02342) is lower than α. Consequently, we may reject the null hypothesis (*H0*).

Table 5.4 Hypothesis test for analysis of English reading.

	Shapiro-Wilk		**Parametric Test**
	Good	**Medium/Poor**	**t-test**
p-value	0.9505	0.9505	0.02342

We also evaluated our results in terms of magnitude, testing the effect size measure. We calculate Cohen's D [91] in order to interpret the size of the difference between the distribution of the groups. We used the classification presented by Cohen [91]: 0 to .1: No Effect; .2 to .4: Small Effect; .5 to .7: Intermediate Effect; .8 and higher: Large Effect.

The magnitude of the result ($d = 0.814$) also confirmed that there is a difference (Large Effect) on the accuracy values with respect to both groups. This evidence reinforces our hypothesis and shows that the results were statistically significant.

In addition, we analyzed the feedback form and we highlight the main notes at the following (translated to English): (i) I had some difficulties to understand and decide about complex comments; (ii) I had the feeling that I needed to know the software context better; (iii) I believe some tips on English comments could help us to interpret the complex comments. This data is aligned with our finding that indicates that English reading skills may affect the task of analyzing code comments to identify TD in software projects.

5.7.2 The impact of the experience level on the TD identification process (RQ2).

In order to investigate the impact of the experience level on the TD identification process, we calculated the accuracy values for each participant with respect to the oracle. We show the accuracy distribution by experience level of the participants in Figure 5.3. From this figure, it is possible to note that the box plots have almost the same level of accuracy regarding high, medium and low experience. Considering the median values, the values are very similar. Participants with high and low experience have the same median value (0.66), whereas the median of participants with medium experience is moderately higher (0.71).

We also calculated the variation coefficient. This coefficient measures the variability in each level – that is, how many in a group are near the median. We found the coefficients of 12.91%, 13.17%, and 13.96%, for high, medium and low experience, respectively. According to the distribution presented by Snedecor and Cochran [92], the coefficients are low, showing that the levels of experience have homogeneous values of accuracy.

Finally, we performed a hypothesis test to analyze the experience variable. We defined the following null and alternative hypotheses:

H0: The experiences of the participants do not affect his or her accuracy with respect to the agreement with the oracle.

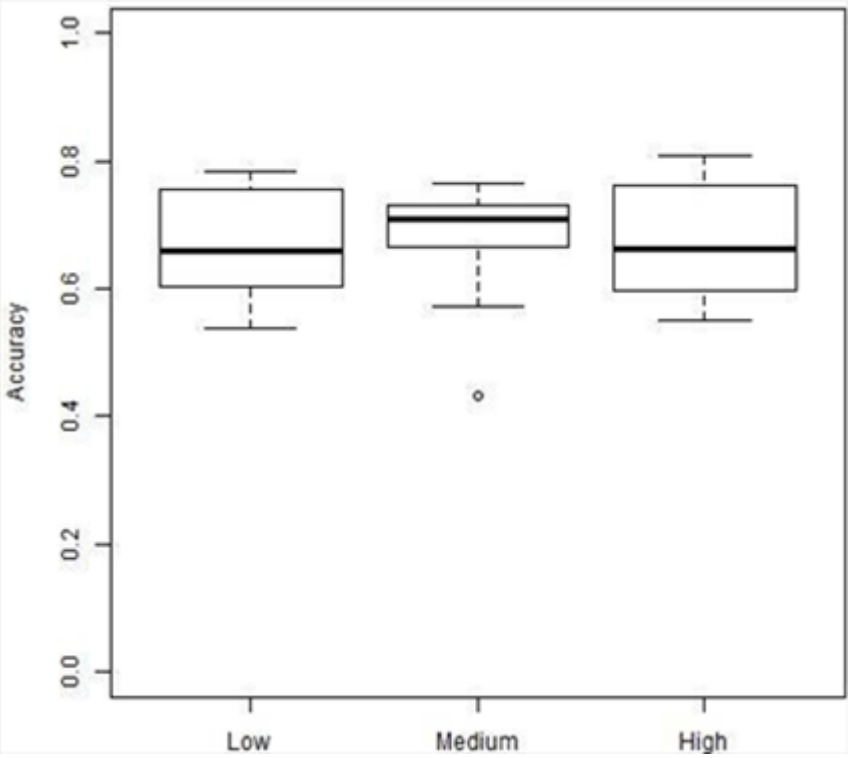


Figure 5.3 Accuracy by participants' experience.

H1: The experiences of the participants affect his or her accuracy with respect to the agreement with the oracle.

After testing normality, we ran Anova, a parametric test to evaluate more than two treatments.

The p-value calculated (p= 0.904) is bigger than the selected α value. In this sense, we do not have evidence to reject the null hypothesis (*H0*).

From the analysis, we consider that the experience level did not impact the distribution of the accuracy values, i.e., when using CVM-TD, experienced and non-experienced participants show the same accuracy when identifying comments that point out to TD items. A possible interpretation of this result is that CVM-TD can be used by non-experienced participants.

5.7.3 The agreement on TD detection (RQ3).

Our analysis considered the number of participants who have chosen a comment. Figure 5.4 shows the ratios in the X-axis and the number of comments in each interval in Y-axis. The ratio values are the proportion of "number of participants who choose the comment" and the "number of participants in the experiment group". For example, a comment from group G1 (the G1 has 11 participants) that was chosen by 10 participants has ratio = 0.91 (that is, 10/11).

We can note that all or almost all participants have chosen some comments as a good

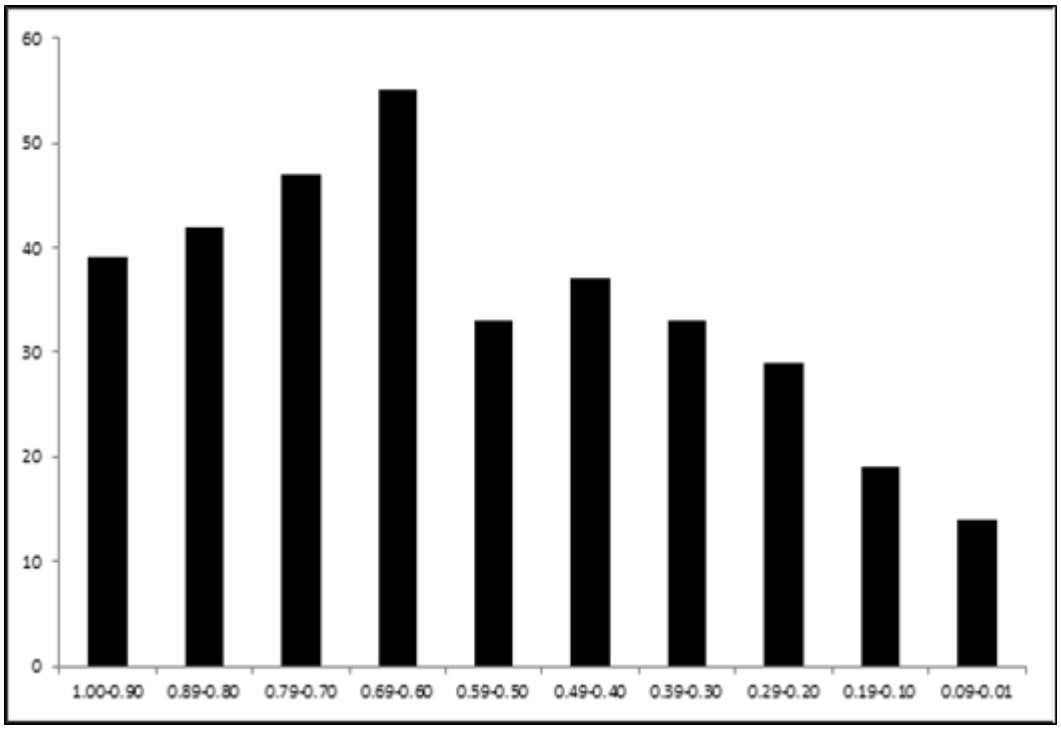


Figure 5.4 Agreement among TD comments.

indicator of TD, which means that these comments had a high level of agreement and CVM-TD filtered comments that may really point out to TD item. Almost 40 comments have ratio intervals between 1 and 0.90. Some examples of such comments are:

```
Example comment #01:
\NOTE: This is temporary and will go away in a "future" release (ratio
= 1)"

Example comment #02:
\// FIXME: this could be a problem...(ratio = 1)"

Example comment #03:
\TODO: Replace the next deprecated call (ratio = 0.90)"

Example comment #04:
\TODO: This functionality needs to be moved someplace useful...(ratio
= 0.90)"
```

The whole set of these comments is available at [28].

On the other hand, considering the agreement among all participants identifying TD comments, we found a low coefficient. We conducted the Finn test to analyze the agreement in each group, considering all comments. Table 5.5 presents the agreement coefficient values. The level of agreement was 'slight' and 'fair' according to [90] classification.

Table 5.5 Finn agreement test.

	Finn	p-value	Classification levels
Group 1	0.151	3.23e-05	Slight
Group 2	0.188	5.74e-07	Slight
Group 3	0.265	8.34e-12	Fair

5.7.4 The impact of CVM-TD and the vocabulary on selection of TD comments (RQ4)

We analyzed the candidate comments identified by the oracle as TD comments. Table 5.6 shows the number of comments identified by the oracle. We observed that almost 60% of comments filtered by patterns that belong to the vocabulary (candidate comments), proposed in [24] and presented in Chapter 3, were identified as good indicators of TD by the oracle. We notice that the percentage of candidate comments chosen as TD comments (a comment that may indicate a TD) is bigger that the percentage of candidate comments chosen in *FindTD I*.

Table 5.6 TD comments identified by the oracle.

	Group 1	Group 2	Group 3
Number of candidate comments	123	124	106
Number of TD comments	71 (57.72%)	83 (66.94%)	58 (54.72%)

Regarding the average accuracy of all participants against the oracle, the overall value is 0.673. On average, this shows that the participants achieved good accuracy values. We also analyzed the dispersion of the set of values. The standard deviation (SD) is 0.087 which is considered a low value. In particular, the low SD indicates that the values do not spread too much around the average. That is, it indicates low dispersion of the accuracy values from the average. The box-plot in Figure 5.5 represents the distribution of accuracy values for all participants. This figure shows that the values have a homogeneous distribution of accuracy.

We can also see the accuracy distribution by each group of the participants in Figure 5.6. From this figure, it is possible to note that the box-plots have almost the same level of accuracy. Considering the median, the values are very similar.

Next, we analyzed the number of TD comments chosen by both participants and the Oracle. Table 5.7 shows the TD comments identified by the Oracle and also by at least 50% of all participants in each group. This means that 66 comments were chosen by the Oracle and by at least 5.5 participants of the group 1. For group 2, 72 comments were chosen by the Oracle and by at least 5 participants. While for group 3, 51 comments were chosen by the Oracle and by at least 5.5 participants. The achieved results indicate that 89.21% of the TD comments that were chosen by the Oracle were also chosen by at least 50% of all participants. The results indicate that the vocabulary helps the participants

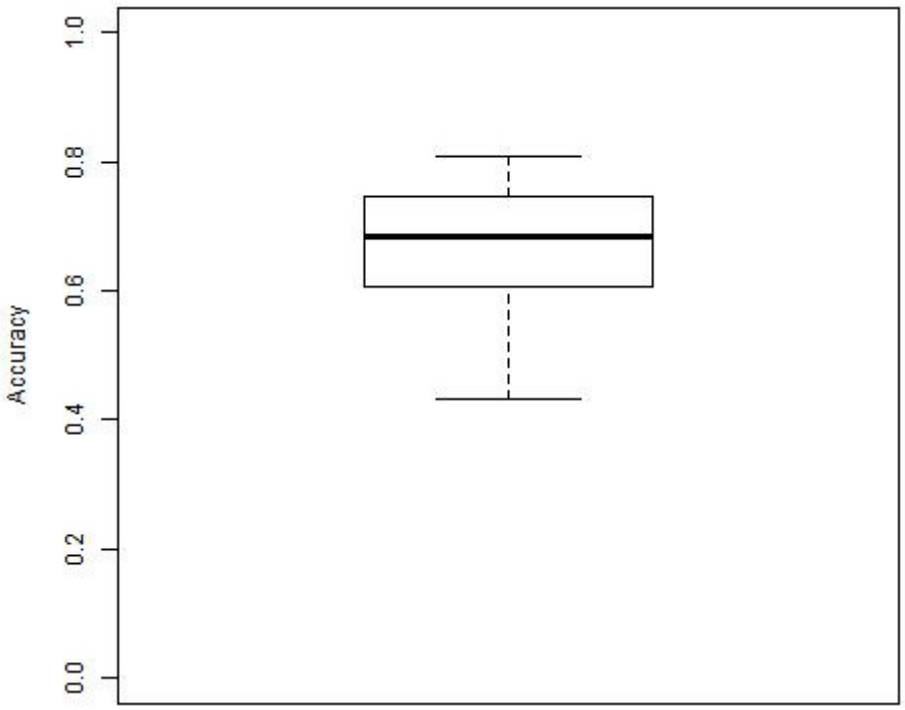


Figure 5.5 The distribution of the accuracy values.

to identify comments that can point out to TD items.

Table 5.7 TD comments identified by the Oracle and participants.

	Group 1	**Group 2**	**Group 3**
Number of candidate comments	123	124	106
Number of TD comments	71 (57.72%)	83 (66.94%)	58 (54.72%)
Number of TD comments chosen by participants and Oracle	66 (92.95%)	72 (86.74%)	51 (87.93%)

We also analyzed the number of comments per rate of participants that chose the comments indicated by the Oracle as a TD comment. Figure 5.7 shows the ratios in the X-axis and the number of comments in each interval in Y-axis. The ratio values are the proportion of "number of participants who choose a comment indicated as TD comment by the Oracle" and the "number of participants in the experiment group". For example, a comment from group G1 indicated as TD comment by the Oracle (G1 has 11 participants) that was chosen by 11 participants has ratio = 1.00 (that is, 11/11).

When looking at the number of comments reporting TD, it is possible to note that, on average, the participants identified many comments that were filtered by the vocabulary and selected as TD indicators by the Oracle. This means that the vocabulary proposed by CVM-TD helped the participants to comprehend and identify comments that may

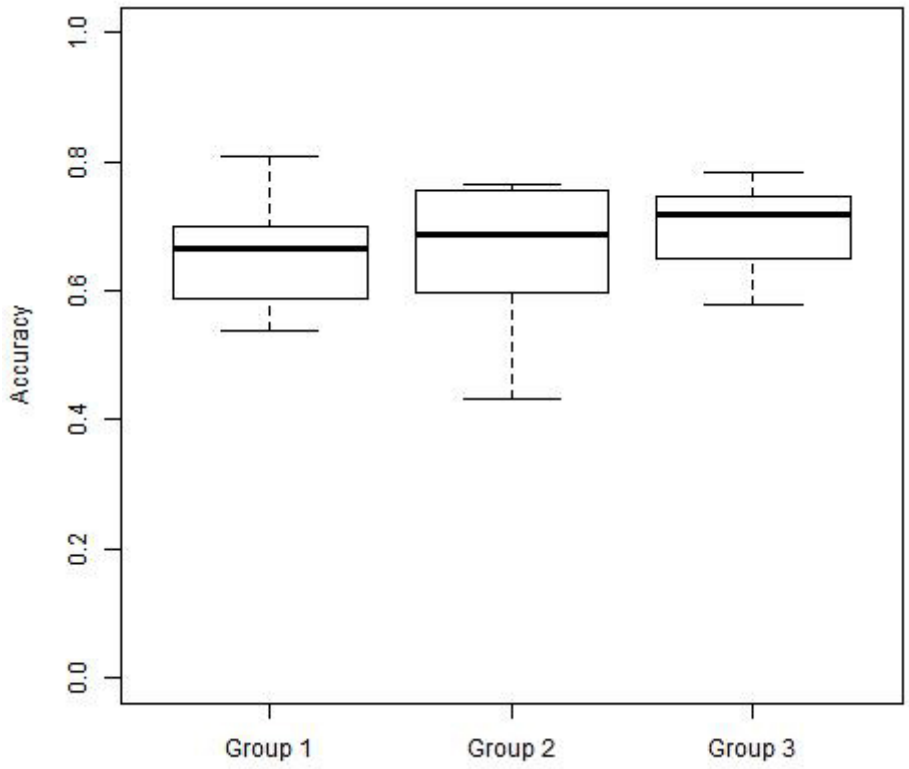


Figure 5.6 Accuracy values by groups.

point out TD items. More than 170 comments have ratio intervals equal or higher than 0.5.

In order to perform a deep analysis, we investigated a sample of these comments. From the analysis, let's consider the following comments:

```
Example comment #05 (ratio 1.00):
/* Install the trap to "eat" SecurityExceptions.  * NOTE: This is
temporary and will go away in a "future" release */
```

The above comment (#5) mentions issues in the source code and highlights that a specific part of code is temporary and needs to be removed in a future release. The comment has three patterns: *"the trap"*, *"NOTE: This is temporary"*, and *"future release"*. In other words, it was indicated as a comment reporting a TD item by Oracle and all participants.

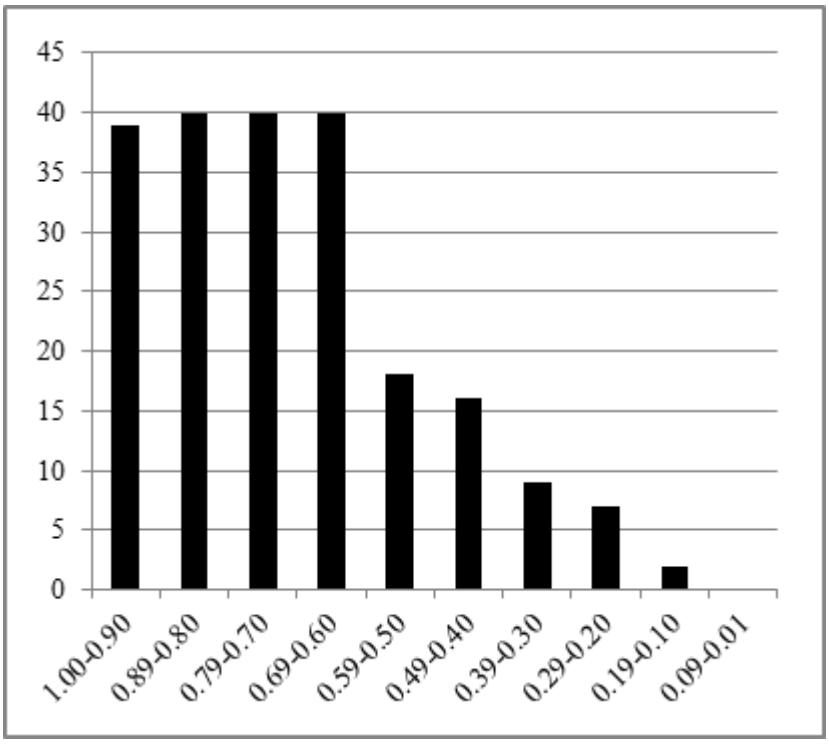


Figure 5.7 Number of comments per rate of participants in accordance with oracle.

```
Example comment #06 (ratio 0.92):
//* TODO: Why is this here?  Who is calling this?
@see java.beans.VetoableChangeListener#vetoableChange
(java.beans.PropertyChangeEvent) */

Example comment #07 (ratio 0.64):
// This is somewhat inconsistent with the design of the constructor
// that receives the root object by argument.  If this is okay // then
there may be no need for a constructor with that argument.
```

In example comments #6 and #7, the developers describe situations correlated with violation of principles of good design, inadequate location of a method, and inconsistency with the design. We can see this context through the patterns "*Why is this here?*", "*Who is calling this?*", and "*inconsistent with the design*". Besides these patterns, we can note the tag "*TODO*" reporting the need to revisit this code in future.

```
Example comment #08 (ratio 0.91):
/* TODO: needs documenting, why synchronized?  */

Example comment #09 (ratio 0.82):
/* TODO: As currently coded, this actually returns all
BehavioralFeatures which are owned by Classifiers contained in the
given namespace, which is slightly different than what's documented.
It will not include any BehavioralFeatures which are part of the
Namespace, but which don't have an owner.  */
```

Comments #8 and #9 are examples that we consider as documentation debt. In the above comment, the developers describe the necessity for documentation and show their worries about the missing or inadequate documentation. Maybe this TD item could not be identified using only code-based metrics because the necessity of documentation cannot be measured only analyzing the source code.

```
Example comment #10 (ratio 0.82):
// The following debug line is now the single most memory consuming
// line in the whole of ArgoUML. It allocates approximately 18% of
// all memory allocated.  // Suggestions for solutions:  // Check
if there is a LOG.debug(String, String) method that can // be used
instead.  // Use two calls.  // For now I (Linus) just comment it
out.
```

Comment #10 reports strategies that caused a very large processing consuming. The patterns "*most memory consuming*" and "*all memory allocated*" may refer to build related issues that make a task harder, and more time/processing consuming unnecessarily.

```
Example comment #11 (ratio 0.91):
/* TODO: Replace the next deprecated call.  This case is complicated
* by the use of parameters.  All other Figs work differently.  */
```

In comment #11, the developers describe situations correlated with bad coding practices, deprecated code, and difficult to be maintained in the future. The patterns "*TODO: replace*", "*deprecated call*", and "*case is complicated*" refer to the problems found in the source code, which can negatively affect the quality of the code making it more difficult to be maintained in the future. They are examples of what we consider as code debt.

```
Example comment #12 (ratio 0.91):
/* TODO: The copy function is not yet completely implemented - so we
will have some exceptions here and there.*/.
```

Comment #12 describes the lack of synchronism between optimal requirements specification and what is currently implemented. The pattern "*is not yet completely implemented*" indicates a requirement that are only partially implemented.

Some of these comments have patterns which may report more clearly a situation related with the description of TD items, such as, "*TODO: this is temporary*", "*inconsistent with the design*", and "*TODO: needs documenting*". We analyzed some of the top comments having ratio higher than 0.8. Table 5.8 lists the patterns identified by the participants in these comments. Besides the fact that these comments have patterns that we considered to clearly identify TD, we can also see that many of them have more than one pattern reporting a TD context. This also helps indicating that the patterns can be used to filter comments describing a situation of TD. We deeply analyze these patterns in *FindTD IV*.

False-positive comments

Although the vocabulary allowed filtering good comments reporting TD, it also returned false positives. The oracle and participants identified 164 comments, which were

Table 5.8 List of patterns identified by participants in these comments.

ID comment	Ratio	Identified Patterns
17	1.00	the trap
		NOTE: This is temporary
		”future” release
945	1.00	This is a bug
1005	1.00	TODO: This is probably not the right location
1064	1.00	FIXME: this could be a problem
26	0.91	TODO:
		Need to be in their own files?
		TODO: Why is this here?
44	0.91	Who is calling this?
		TODO: needs documenting
68	0.91	a temporary solution
122	0.91	Should be fixed
318	0.91	TODO: split into
485	0.91	problems directly in this class
		Critic problem
		TODO: Replace
521	0.91	Deprecated
		This case is complicated
		Is there no better way?
549	0.91	TODO: implement this
835	0.91	TODO : ??
1227	0.91	code is not used
		Why is it here?
		causes dependency

returned by the vocabulary, as comments that do not report a situation of TD, that is, they are false positives. This shows that 46.46% of the comments identified by our approach may have been misclassified by patterns of the vocabulary (353 comments were returned in total).

Some examples of these false positives follow:

```
Example comment #13:
// NOTE: This is package scope to force callers to use
ResourceLoaderWrapper

Example comment #14:
// Note that this will not preserve empty lines // in the body
```

The *eXcomment* selected these candidate comments (#13 and #14) because of presence of the "*NOTE:*" but the comment only gives an overview of the information about the code implementation.

```
Example comment #15:
/* Now we have to see if any state in any statemachine of * classifier
is named [name].  If so, then we only have to link the state to c.
*/
```

The pattern "*have to*" was interpreted as an open task verb reporting a task to be done but it only composes an expression describing the functionality of the code.

```
Example comment #16:
/* This is used in the todo panel, when "By Poster" is chosen for a *
manually created todo item.*/
```

The pattern "*TODO*" made our tool and vocabulary selecting the comment as a candidate comment, while the term was used only to describe the name of a component.

```
Example comment #17:
/* This next line fixes issue 4276:  */
```

The pattern "*fixes*" may suggest something to be fixed in the project but, in this case, the comment describes the opposite, a piece of the code fixing an issue.

This result indicates that the vocabulary and *eXcomment* need to be better calibrated to reduce the number of returned comments that do not report a situation of TD. With this in mind, we performed a complementary investigation with the purpose of investigating false-positive comments and identifying the patterns that are more decisive to detect TD items in *FindTD III* and *FindTD IV*.

5.7.5 The most chosen code comment patterns by participants (RQ5).

Table 5.9 shows the top 20 selected patterns. Column "*Patterns*" lists the patterns identified by participants, "*# of occurrences*" summarizes the number of times that a participant highlighted the patterns as important, and "*# of Participants*" summarizes the total of participants who have chosen the pattern as important to identify a TD item.

The most selected pattern was '*Todo*". It was chosen by 26 of 32 participants (81.25%), followed by "*need to*" selected by 22 participants (65.63%), "*remove*", and "*Todo: Replace*" by 16 participants (50%). Moreover, several other patterns have the tag "*Todo*", such as "*Todo: This does not work!*" and "*Todo: Is this needed/correct?*". The pattern "*Todo*" was identified as a decisive pattern in 899 cases by the participants. This highlights the importance of "*Todo*" for composing important patterns to identify TD

through code comment analysis. Besides the pattern "*Todo*", the patterns "*need to*", "*remove*", "*replace*", "*Is this needed?*", "*must be*", "*should be*", and "*does not work*" were also chosen as decisive patterns by the participants. The complete list is publicly available [28].

Even though these results show that there are decisive patterns to identify a situation of TD, further investigations are needed for analyzing how much each pattern is decisive to identify TD. In this way, we performed another experiment (*FindTD III*) to widely analyze and classify all patterns by considering the importance level of the patterns to point out to a TD item.

Table 5.9 Top 20 patterns more chosen by participants as decisive to indicate a TD item.

Patterns	# of occurrences	# of Participants
TODO	899	26
need to	48	22
Remove	54	19
?	49	19
Replace	19	14
Is this needed?	15	14
must be	22	13
should be	20	12
does not work!	18	12
Critic	21	11
Check	14	11
Fix	22	11
Why?	21	10
cyclic dependency	10	10
this is a bug	10	10
Note	13	9
Move	13	9
have to	12	9
Temporary	9	9
probably redundant	9	9

5.8 SUMMARY OF FINDINGS AND INSIGHTS

Our results suggest that the English reading level of the participants may impact the identification of TD through comment analysis. Participants with good English reading skills had accuracy values better than participants who have medium/poor English read-

ing skills. On the other hand, participants with poor/medium English profile were able to identify a good amount of TD comments filtered by the contextualized vocabulary.

We also observed in the feedback analysis that some participants had difficulties to understand and interpret complex comments, and tips might help them with this task. Our assumption is that some tips may support developers to make a decision on the TD identification process. For instance, highlight the TD terms or patterns of comment from the contextualized vocabulary into the comments.

Considering the impact of experience on TD identification, we could not conclude that the experience level affects the accuracy. This evidence can indicate that comments selected by the vocabulary may be understood by either experienced or non-experienced observers. This reinforces the idea that the TD metaphor aids discussion by providing a familiar framework and vocabulary that may be easily understood [93] [34].

Considering the agreement among participants, the results revealed some comments pointed out as a good indicator of TD, with a high level of agreement. It may evidence the contribution of CVM-TD as a support tool for the TD identification. However, in general, the level of agreement among participants was considered weak. We believe that this occurred due to a lot of comments to be analyzed, and many comments selected by the contextualized vocabulary that does not indicate a TD item. In this way, the level of agreement might rise whether the vocabulary is more accurate.

We notice that a high number of comments filtered by the CVM-TD was considered to indicate TD items. Given the contribution of the CVM-TD to support TD identification, the results suggest that many comments that were chosen by Oracle were also selected by participants as comments that may point out to a TD item. This means that the vocabulary proposed by CVM-TD helped the participants to comprehend and identify comments that may report a situation of TD.

These results provide preliminary indications that CVM-TD and the contextualized vocabulary can be considered a valuable support tool to identify TD item through code comments analysis. Different from code metrics-based tools, code comments analysis allows us to examine human factors to explore developers' point of view and complement the TD identification with more contextual and qualitative data. Both approaches may contribute to each other to make the automated tools more efficient.

The last aspect we analyzed was the patterns chosen by participants as decisive to indicate a TD item. We identified a list of the top 20 patterns. We could note that there are patterns that are more decisive than others, but we need to perform further investigations on the patterns considering their importance level to identify a TD item. To do that, we performed another controlled experiment (*FindTD III*) to characterize the CVM-TD and the vocabulary with the purpose of identifying the most important patterns, and the relationship between patterns and TD types.

5.9 THREATS TO VALIDITY

We followed the checklist provided by [94] to discuss the relevant threats to this controlled experiment.

5.9.1 Construct Validity.

To minimize the mono-method bias, we used an accuracy and agreement test to provide an indication of the TD identification through comment analysis. We selected the researchers that composed the Oracle. In order to mitigate the biased judgment on the Oracle, its definition was performed by three different researchers with knowledge in TD. Two of them selected the TD comments, and the third researcher did a consensus to decrease the bias. Finally, to reduce social threats due to evaluation apprehension, participants were not evaluated.

Another threat involves the definition of the proposed vocabulary. It is possible that the set of patterns and combinations used by our model and vocabulary are simply too many to be studied. An alternative would be to limit the studies to specific contexts and software domains. Another point is related to the imprecision in the filtering of candidate comments. Our results might be affected by false positives and false negatives returned by the vocabulary and the automatic filtering heuristic. To reduce this threat in the next experiment, we intend to calibrate the vocabulary and *eXcomment* to decrease the number of comments that do not report a situation of TD.

5.9.2 Internal Validity.

The first internal threat we have to consider is the subject selection since we have chosen all participants through a convenience sample. We minimized this threat organizing the participants in different treatment groups divided by experience level.

Another threat is that participants might be affected negatively by boredom and tiredness. To mitigate this risk, we performed a pilot study to calibrate the time and amount of comments to be analyzed. To avoid the communication among participants, two researchers observed the operation of the experiment at all times. A further validity threat is the instrumentation, which is the effect caused by artifacts used for the experiment. Each group had a particular set of comments, but all participants used the same data collection form format. To investigate the impact of this threat in our results, we analyzed the average accuracy in each group. Group G1 has an average value equal to 0.65. For group G2, the average value is equal to 0.66, and group G3 is equal to 0.69. From these data, it is possible to note that groups have almost the same level of average accuracy. It means that this threat did not affect the results.

5.9.3 External Validity.

This threat relates to the generalization of the findings and their applicability to industrial practices. This threat is always present in experiments with students as participants. Our selected samples contained participants with different levels of experience. All participants have some professional expertise in the software development process. It is an important aspect in mitigating the threat. A further threat is the use of software that may not be representative for industrial practice. We used software adopted in the practice of software development as an experimental object to mitigate the threat.

5.9.4 Conclusion Validity.

To avoid the violation of assumptions, we used normality test, Shapiro-Wilk, and a parametric test, the *t-test*, for data analysis. To reduce the impact of the reliability of treatment implementation, we followed the same experimental setup on both cases.

Chapter

6

This chapter presents the third study of the family of experiments that we carried out as a controlled experiment. We performed the FindTD III from insights of FindTD II, by changing the setup and controlling other variables. We carried out a qualitative analysis to improve the model and the vocabulary, identifying the most significant patterns and the relationship between patterns and the types of TD.

FINDTD III

6.1 GOAL OF STUDIES AND RESEARCH QUESTIONS (RQ)

This study aims to **characterize** the CVM-TD and the vocabulary **with the purpose** of identifying the most important patterns, and the relationship between comment patterns and TD types, **with respect to** score calculated when evaluating patterns **from the point of view of** the researcher **in the context of** professors and software engineering master and PhD students with professional experience analyzing comment patterns identified and classified in previous experiment.

The experiment will provide us a new release of the contextualized vocabulary. Our assumption is that this new release will be more contextualized and accurate because it was created considering the opinion of participants of the *FindTD II* and *FindTD III*. After this experiment, we also intend to apply the results from this experiment to develop new features in the *eXcomment*. They will be associated with the new vocabulary to quickly support the interpretation of comments.

To achieve the overall goal, we defined five RQs:

RQ1. *Which comment patterns identified in FindTD II are decisive to identify TD?*

With this question, we intend to investigate how much a pattern is decisive to point to a TD. To evaluate this question, we consider the patterns chosen by participants from FindTD II in order to identify how much each pattern can be decisive to identify a TD item. For that, we used the score medians measured in this experiment [95]. We hope to provide a more confident set of patterns as a result from this RQ.

RQ2. *Is it possible to improve/evolve CVM-TD and the vocabulary through new experiments?*

This RQ aims to analyze the evolution of both model and vocabulary through three releases of the vocabulary. The first was generated from the model and applied in *FindTD*

I, the second was generated from the *FindTD I* and applied in *FindTD II*, and the third was generated from *FindTD II* and analyzed in *FindTD III*.

RQ3. *What are TD themes identified from the set of patterns?*

This question aims at organizing and grouping similar data from a set of patterns into categories or themes related to TD contents. Consequently, we can link different patterns to the classified themes. As a result of this question we hope to provide a list of themes related to TD context. We will use the term "TD themes" to reference the themes related to TD context in this work, for example the theme *Open Task* is related to tasks that need to be done.

RQ4. *Which TD themes can be associated with TD indicators proposed by [13]?*

Considering the set of TD themes provided by RQ3, we intend to investigate which themes and patterns have a relationship with TD indicators. From this question, we hope to provide a classification of themes related to TD indicators.

RQ5. *Which types of TD exist in the studied software projects?*

This question aims to analyze the relationship provided by previous RQs to investigate what types of TD can be detected though code comments analysis in the studied projects. In particular, we focus on the detection and quantification of the types presented in Chapter 2, Section 2.1.2.

Figure 6.1 summarizes the process of this study. First, we selected the artifacts and 25 participants for the controlled experiment. Next, participants analyzed the patterns and registered the level of importance for each pattern of the vocabulary to identify TD. As the third step, participants performed a consensus step to mitigate issues on the judgment of each pattern. In the last step, a quantitative and qualitative analysis were performed in order to analyze the research questions.

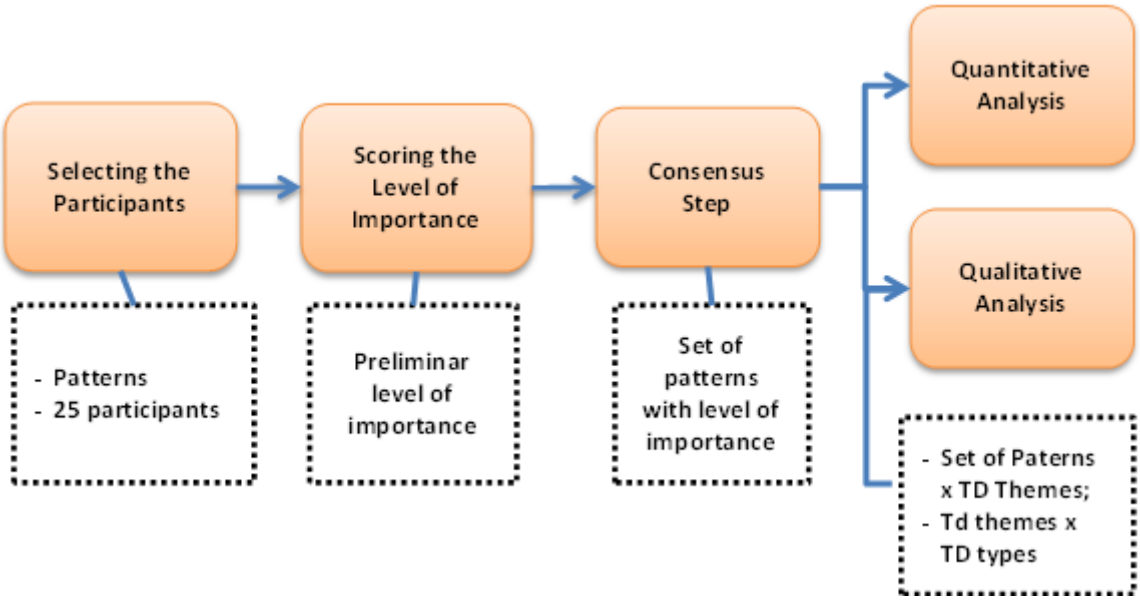


Figure 6.1 Process of FindTD III.

6.2 PARTICIPANTS

The participants of the study was selected using convenience sampling [20]. Our sample consists of 25 participants, where 2 are professors at the Federal University of Bahia, 4 are software engineering PhD students, and 19 are software engineering master students.

In order to classify the profile of the participants and their experience in the software development process, we adopted the same questionnaire applied in *FindTD II*. The results showed that most of participants had high experience level.

The participants were classified using the same strategy of the FindTD II, which was presented in the Section 5.2. We had 22 participants with high experience level, two with medium experience, and one with low experience. The participants were randomly divided into ten groups. Half of the group had two participants and for the other half we had three participants. The design was done in such a manner that each group analyzed a different set of patterns and each pattern was analyzed by at least two participants. We adopted this plan in order to avoid an excessive number of patterns to be analyzed by each participant and in the same time it permits that a set of patterns to be analyzed by more than one participant. Table 6.1 shows the distribution of the participants.

Table 6.1 Distribution of the participants among groups in *FindTD III*.

Group	Participants by experience level		
	High	Medium	Low
G1 (3)	2	1	-
G2 (3)	2	-	1
G3 (3)	3	-	-
G4 (2)	2	-	-
G5 (3)	3	-	-
G6 (3)	3	-	-
G7 (2)	2	-	-
G8 (2)	2	-	-
G9 (2)	1	1	-
G10 (2)	2	-	-
Total (25)	**22**	**2**	**1**

6.3 INSTRUMENTATION

6.3.1 Forms.

The experimental package is available at [29]. We used the same slides of the *FindTD II* for the training and four forms to perform the experiment. The consent, characterization, and feedback forms were the same applied in *FindTD II*. Thus, here, we detail only the data collect form.

Data collect form: contains a list of patterns. During the experiment, the participants were asked to analyze previously selected pattern, considering how much they report a situation of TD.

6.3.2 Software Artifact and Candidate Comments.

We used the patterns of our contextualized vocabulary and the patterns selected from TD comments in *FindTD II*. As was mentioned before, in *FindTD II* the participants highlighted the chunks of text that were decisive for choosing those TD comments.

A total of 349 patterns were selected to be analyzed in *FindTD III*. Each pattern chosen in *FindTD II* was related to its comments and listed in the data collect form in the same order in which they are in the code.

6.4 ANALYSIS PROCEDURE

We used quantitative and qualitative analysis by considering different perspectives to evaluate the five RQs. We consider all comment patterns chosen by experiment participants in *FindTD II* in order to address the RQs. To do that, we considered three perspectives:

Score median measures: We used the score medians in order to answer the RQ1. The scores represent the importance level of the pattern for pointing out to a TD item. We used two approaches to address the research question. The first considered the score medians of the most selected patterns by participants in *FindTD II*, and the second ran an analysis of all patterns.

Releases of the vocabulary: To analyze the evolution of model and vocabulary, RQ2, we considered three releases of the vocabulary. Each release was generated from the model, *FindTD I* and *FindTD II*.

Coding techniques: We designed and conducted a qualitative study to deeply investigate the patterns in order to evaluate the questions RQ3, RQ4, and RQ5. To do that, we considered the last release of the vocabulary.

6.5 PILOT STUDY

We carried out a pilot study using the same setup of the pilot performed in *FindTD II*. The pilot took 1 hour and was carried out in a Lab at the Federal University of Bahia (Bahia-Brazil). The student analyzed and classified 70 patterns.

The pilot helped us to evaluate the use of the data collection form, the necessary time to accomplish the task and, mainly, the number of patterns used by each group in the experiment.

6.6 OPERATION

The experiment was conducted in a classroom at the Federal University of Bahia. The operation of the experiment also was divided into different sessions. The training and experiment itself were performed at the same day. For training, we performed the same

presentation performed in *FindTD II*. The participants filled the consent and characterization form after the training.

Next, each participant analyzed a set of patterns. The order of the patterns was assigned randomly to each group, so that each participant analyzed almost 40 patterns. This analysis was carried out in the same room where the training was provided. They filled the data collection form pointing out the initial and end time of the task. For each pattern listed in the form, the participants scored the level of importance for each previously selected pattern. The participants used an ordinal scale of zero to four to indicate the levels. Each pattern was analyzed by at least two participants, ensuring different opinions about it. After that, a consensus step was performed to mitigate the different opinions.

As in *FindTD II*, the participants were asked to not discuss their answers with others before the agreement step. When they finished, they filled the feedback questionnaire. A total of two hours were planned for the experiment training and execution.

6.6.1 Deviations from the Plan.

We eliminated two participants of our study because they did not complete all the experimental sessions, since we needed all the information to a complete analysis (characterization, data collection, and feedback).

Table 6.2 presents the final distribution of the participants in *FindTD III*. The value in parentheses indicates the final number of participants in each group. In each of the groups G1 and G2, a participant was excluded because of not filling all forms completely.

Table 6.2 Final distribution of the participants among groups in *FindTD III*.

Group	Participants by experience level		
	High	Medium	Low
G1 (2)	2	-	-
G2 (2)	1	-	1
G3 (3)	3	-	-
G4 (2)	2	-	-
G5 (3)	3	-	-
G6 (3)	3	-	-
G7 (2)	2	-	-
G8 (2)	2	-	-
G9 (2)	1	1	-
G10 (2)	2	-	-
Total (23)	**21**	**1**	**1**

6.7 ANALYSIS OF FINDTD III

In this Subsection we describe the results in order to address the research questions presented above.

6.7.1 Analysis of patterns identified in FindTD II (RQ1).

To evaluate this question, we consider patterns chosen by experiment participants from FindTD II in order to identify how much each comment pattern can be decisive to identify a TD item.

The scores represent the importance of the patterns to point out to a TD item. To analyze and identify the decisive patterns, we consider they as: (i) **very decisive pattern** those that were indicated with score medians between 3.5 and 4 (strong patterns), (ii) **decisive pattern** those that were indicated with score medians between 2.5 and 3.4 (strong patterns), (iii) **not much decisive pattern** those that were indicated with score medians between 1.5 and 2.4 (weak patterns), (iv) a pattern that **can be decisive** those that were indicated with score medians between 0.5 and 1.4 (weak patterns), and (v) a **not decisive pattern** those that were indicated with score medians between 0 and 0.4.

We used two approaches to address the research question. The first considered the score medians of the most selected patterns by participants in *FindTD II*. In this case, we intended to verify whether the selected patterns are really considered important for identifying TD. The second ran an analysis of all patterns. In this analysis, we aimed to have an overall view of the importance of all patterns for the identification of TD.

The most selected patterns by participants in *FindTD II*. Table 6.3 shows the top 25 patterns by considering the amount of participants. Column "Selected Patterns" lists the patterns identified by *FindTD II*, "# of Participants" summarizes the total of participants who chosen the patterns as important to identify a TD item in the *FindTD II*, and "Score Medians" shows the medians calculated for each pattern in this analysis.

The patterns "*Todo*" was the most selected pattern. It was chosen by 26 of 32 participants (81.25%) in *FindTD II*, followed by "*need to*" selected by 22 participants (65.63%), "*Remove*" by 19 participants (59.38%), and "*Todo: Replace*" by 14 participants (43.75%). From the table we can also see that several comment patterns have the tag "*Todo*", such as "*Todo: does not work!*" and "*Todo: Is this needed?*". All patterns having the tag "*Todo*" were indicated as decisive or very decisive to identify a TD item. This evidence shows the importance of "*Todo*" for composing decisive patterns to identify TD through code comment analysis.

Besides, 16 patterns were indicated as "decisive" or "very decisive" (64%) to detect a TD idem and only 6 patterns were indicated as "not much decisive" (24%). The patterns "*note, remove, and cause exception*" was indicated as a pattern that "can be decisive" to identify a TD item (12%). No patterns were indicated as "not decisive". The whole list is available at [29].

Analyzing all patterns. Regarding all patterns identified in the *FindTD II* (349) and their score medians calculated here, we found 184 patterns identified as "decisive"

Table 6.3 Top 25 comment patterns by considering the number of participants.

Patterns Selected in FindTD II	# of Partic. (FindTD II)	Score Medians (FindTD III)	Importance Level of Patterns
TODO	26	2.00	Not much decisive pattern
Need to	22	2.50	Decisive pattern
Remove	19	1.00	Can be decisive
Todo: Replace	14	3.5	Very decisive
TODO: Is this needed?	14	3.00	Decisive pattern
Must be	13	2.00	Not much decisive pattern
Should be	12	2.50	Decisive pattern
Todo: does not work	12	4.00	Very Decisive
critic	11	2.00	Not much decisive pattern
Todo: Check	11	4.00	Very decisive
fix	10	2.00	Not much decisive pattern
Why?	10	2.00	Not much decisive pattern
cyclic dependency	10	3.00	Decisive pattern
this is a bug	10	4.00	Very decisive
Note	9	1.25	Can be decisive
have to	9	2.60	Decisive
Todo: temporary	9	4.00	Very decisive
probably redundant	9	3.00	Decisive
should not be needed	8	3.00	Decisive
Todo: implement this!	8	3.00	Decisive
Deprecated	8	3.50	Very decisive
FIXME	8	2.50	Decisive
cause exception	8	1.00	Can be decisive
Todo: Why is this here?	7	4.00	Very Decisive
Remove duplicates	7	2.00	Not much decisive pattern

or "very decisive" (52.72%), 101 patterns identified as "not much decisive" (28.94%), 47 indicated as a pattern that "can be decisive" (13.47%), and 16 identified as "not decisive" (4.58%) to identify a TD idem. The whole list consisting of 349 patterns sorted by score is publicly available [29].

By analyzing this list, we identified 54 patterns that appeared isolated in the comments and 295 patterns that appeared combined with other patterns. In this work, we call them as isolated and composed patterns, respectively. We consider as isolated patterns those that are composed by only a dimension of our model, for example, "*have to*" that is classified as an open task verb or "future" that is classified as an adverb. We consider as composed patterns those that are constituted by a combination of two or more categories, that is, NPs and Semantic Expressions, for example, "*have to move*" that is composed by an open task verb (have to) and an action verb (move). In general, patterns generated by a semantic composition of words may improve the power of contextualization of the comments to identify a TD item. Composed patterns are phrases or related words that provide high context information.

We counted 165 (55.93%) composed patterns scored as "very decisive" or "decisive", and only 19 isolated patterns (35.19%) that were chosen as "very decisive" to identify a TD item. In this sense, composed patterns seem to be more contextualized and decisive than isolated patterns to identify TD.

Moving on, we identified some isolated patterns that were scored as "not much decisive" or as "can be decisive". However, when they appeared combined with other patterns (composed patterns), their scores tend to be higher than isolated patterns. In this context, some patterns only make sense when they are combined with other ones. Table 6.4 presents in detail some of these cases. On the left side, we illustrate some isolated patterns, their scores, and their importance level to detect TD items. On the right side, we illustrate the composed patterns related to the isolated pattern, their scores, and their importance level. Only the pattern "*will be replaced*" broke this general trend.

We did not list the case of the tags "*Todo, Note and FIXME*" on Table 6.4 because of many patterns combined with them. There are 90 (23.21%) patterns composed by tags, for examples, "*Todo: We should have*", "*Todo: This does not work!*", "*Todo: needs documenting", and "Todo: not implemented*". The whole list consisting of 81 patterns sorted by score is publicly available [28, 29] From this list, we found 61 patterns identified as "decisive" or "very decisive" (75.31%), 13 patterns identified as "not much decisive" (16.05%), 7 indicated as a pattern that "can be decisive" (8.64%), and none of them were identified as "not decisive" to identify a TD idem. This piece of evidence highlights the fact these tags are important to compose decisive patterns aiming to report a situation of TD.

On the other hand, we identified patterns that were scored as "very decisive" or as "decisive" when appeared isolated and combined with other ones. Table 6.5 presents some patterns that were considered decisive in both situations. In spite of the data shows that composed patterns are considered, in general, terms more contextualized than isolated patterns to identify a TD item, there are rare cases where the patterns provide high level of TD context regardless of whether they are generated by a semantic composition or composed by only a category of words.

Table 6.4 Composed patterns were scored better than isolated patterns.

Patterns Alone	**Importance Level of Patterns**	**Composed Patterns**	**Importance Level of Patterns**
Is needed	Not much decisive	Todo: Is this needed?	**Very decisive**
		Todo: Is this needed/correct?	**Decisive**
Fail	Not much decisive	Todo: Why does it fail	**Very decisive**
		assert fails	**Decisive**
		fails the test	**Decisive**
		complete fail	**Decisive**
Replace	Not much decisive	replace the above deprecated	**Very decisive**
		Todo: Replace	**Very decisive**
		will be replaced	Not much decisive
Fix	Not much decisive	Dirty fix	**Very decisive**
		this needs to be fixed	**Very decisive**
		should be fixed	**Very decisive**
		Todo: Fix	**Very decisive**
		no faults are fixed	**Decisive**
Remove	Not much decisive	can remove	Not much decisive
		May remove	Not much decisive
		now remove the pool	**Very decisive**
		Remove duplicates	Not much decisive
		Remove literal	Not much decisive
		Remove the dependente	**Decisive**
		tried to remove	**Decisive**
Could be	Can be decisive	FIXME: this could be a problem...	**Decisive**
		alternative could be	Not much decisive
To do	Can be decisive	Todo: Not sure we need to do this	**Very decisive**
		What to do	**Decisive**
		Find a better way to do	**Decisive**
		Todo: Why do I need to do this?	**Decisive**
		need to do anything here?	Not much decisive
		I chose not to do this	Not much decisive
Check	Can be decisive	Todo: Check!	**Very decisive**
		Todo: Check the name	**Very decisive**
		need to check	Not much decisive
Future	Can bedecisive	Future enhancement	**Decisive**
		Future release	**Decisive**
		future improvement	**Decisive**

Table 6.5 Both isolated and composed patterns were sored as "very decisive" or "decisive" patterns.

Alone Patterns	Score Avg	Importance Level of Patterns	Composed Patterns	Score Avg	Importance Level of Patterns
Required	3.00	**Decisive**	Information that is required	3.00	**Decisive**
			Todo: Is this required?	3.00	**Decisive**
Deprecated	3.00	**Decisive**	replace the above deprecated	4.00	**Very decisive**
			deprecated call	3.00	**Decisive**
Error	3.00	**Decisive**	contains some errors	4.00	**Very decisive**
			It 's an error	4.00	**Very decisive**
temporary	4.00	**Very decisive**	temporary solution	4.00	**Very decisive**
			Note: This is temporary	4.00	**Very decisive**
			Todo: This is a temporary method	4.00	**Very decisive**
			Todo: temporary	4.00	**Very decisive**
need to be	4.00	**Very decisive**	need to be in their own files?	4.00	**Very decisive**
			need to be moved someplace useful	4.00	**Very decisive**
			Need to be amended	4.00	**Very decisive**
Have to	2.60	**Decisive**	have to use	3.00	**Decisive**
			have to delete it later	3.00	**Decisive**
			TODO: Do we really have to test	3.00	**Decisive**

We also analyzed the combinations of word classes, code tags, and expressions composing each pattern, for example, in the comment pattern "*Todo: This is a temporary method*" we have a tag (*Todo*), a link verb (*is*), an adjective (*temporary*), and a noun (*method*). In this way, we found 83 different types of combinations. The most used were "open task verbs + action verbs" (e.g. *should be fixed*), "tag + action verbs" (e.g. T*odo: move to*), and "adjective + noun" (e.g. *temporary solution*). In addition, 25 combinations (30.12%) are compound by a tag, and on the whole, patterns assembled with a tag were scored as decisive to identify a TD item, this case was observed in several comments. This also highlights the importance of tags to support the TD identification process. The whole list of the all combinations is available at [29].

In order to gather further detail, we carried out a qualitative analysis in a sample of comments that were chosen as good indicator of TD by all or almost all participants from *FindTD II*, as can be seen in subsection 5.7.3. We chose these comments because they had high level of agreement concerning TD contents. Table 6.6 presents some of these comments, the patterns recognized for each comment, and their score medians. From this analysis, we observed that almost all comments have at least one patterns selected as "decisive" or "very decisive" to identify TD, such as "*Future release*" and "*dependency cycle*". Decisive patterns may point to important TD indicators and help developers to comprehend a TD context described into code comments.

In addition, comments composed by decisive patterns were indicated as a good TD indicator. It means that when the number or quality of patterns embedded into comment is high, the possibility of the comment is describing a TD is also high. Consequently, this observed trend can set thresholds in order to select comments by considering level of contextualized information, for example, if a comment has very decisive or decisive patterns, there is a high possibility that it describes a TD context.

6.7.2 Evolution of CVM-TD and its Vocabulary (RQ2)

CVM-TD is a contextualized structure of patterns that focuses on using of word classes and code tags to provide a contextualized vocabulary on TD. Thus, through the relationships proposed by CVM-TD, patterns were systematically related with each other, resulting in a first set of composed patterns (vocabulary) [24], as presented in Chapter 3. To analyze the evolution of the model and the vocabulary, we considered three releases of the vocabulary. The first was generated from the model and applied in *FindTD I*, the second was generated from the *FindTD I* and applied in *FindTD II*, and the third was generated from *FindTD II* and analyzed in *FindTD III*.

Table 6.7 presents the number of word classes, code tags, expressions, and NPs by each release. The complete releases of the vocabulary are available at [29]. In the first release, the vocabulary was composed of 128 patterns and all of them were proposed by CVM-TD. The second had 255 patterns. In this release, 127 different patterns were inserted through FindTD I. Whereas the third and last release received 349 new patterns in the FindTD II, totalizing 604 patterns. These patterns provide a growth of the vocabulary and its contextualization.

We can note that the number of patterns has increased 471.88% over the experiments.

Table 6.6 Some comments with high agreement among participants in FindTD II.

Comments Selected as Important	Comment Patterns	Score Avg	Import. Level
/**__Strange__, but the Link.getConnection() *returns a Collection, not a List! *__This is a bug__, compared to the UML standard *(IMHO, mvw). Hence,the LinkEnd is added to the end instead... */	Strange	2.00	Not much decisive
	This is a bug	4.00	**Very decisive**
/* Install __the trap__ to "eat" SecurityExceptions. * * __NOTE: This is temporary__ and will go away in a __"future" release__ *...	The trap	4.00	**Very decisive**
	Note: This is temporary	4.00	**Very decisive**
	Future release	3.00	**Decisive**
/* ... **TODO: Which is best? Is there any other way?** */	Todo: Which is best?	3.00	**Decisive**
	Is there any other way?	2.00	Not much decisive
/* /* __TODO__: This code manually processes the ElementImports of a * Package, but we __need to check__ whether MDR already does something * similar automatically as part of its namespace processing. */	Todo	2.00	Not much decisive
	need to check	1.00	Can be decisive
// __TODO__: This class is part of a __dependency cycle__ with ProjectBrowser and // GenericArgoMenuBar, but __should be fixed__ if project open/close is moved // out of ProjectBrowser.	Todo	2.00	Not much decisive
	should be fixed	4.00	**Very decisive**
	dependency cycle	4.00	**Very decisive**
/* TODO: Is this needed? */	Todo: Is this needed?	4.00	**Very decisive**
/* **TODO: needs documenting, why synchronized?** */	Todo: needs documenting	4.00	**Very decisive**
	why synchronized?	2.00	Not much decisive

Table 6.7 Evolution of the model and vocabulary.

Categories of the Model	1° Release	2° Release	3° Release
Tags	10	10	10
Expressions and NPs	30	97	361
Action and Edit Verbs	12	46	67
Open Task Verbs	13	15	23
Modal Verbs	9	9	10
Adjective	44	65	75
Adverbs	3	5	31
Noun	7	7	14
Symbol	-	-	5
Acronym and Abbreviations	-	-	8
Total	**128**	**255**	**604**

The last release is significantly bigger than the other ones. Figure 6.2 shows that all categories increased, except tags. The most increased category was Expressions and NPs.

We also observed that the categories Symbols, Acronym and Abbreviations emerged in the last release. These new categories can be seen into dashed boxes in Figure 6.3.

The results also show that the candidate comments filtered by the second release are more accurate than those by first release. This indicates that a vocabulary more contextualized can better filter the candidate comments returning comments more contextualized and related to TD context. We intend to continue comparing the third release of the vocabulary with others in FindTD IV.

6.7.3 A qualitative analysis on comments patterns (RQ3, RQ4, and RQ5)

To evaluate the questions RQ3, RQ4, and RQ5, we designed and conducted a qualitative analysis to investigate the patterns. We used the coding techniques mentioned in Chapter 1, Section 1.4, to extract specific segments of text patterns in order to group them through contextual information. The chunks of data extracted from patterns were translated into themes related to TD contents, which we call TD themes. To do this we coded all patterns from the last release of the vocabulary using the tool OpenCode. OpenCode allows the researcher to apply an unlimited number of codes to specific pieces of text, which may then be reorganized according to specific themes or sets of codes [96]. The purpose was to tag the patterns, group them in a specific set of data, and then uncover relations between them and TD context. Following, we analyzed occurrences of patterns in three OSS projects (ArgoUML, jEdit, and Lucene).

We used two approaches to address the RQ3. The first considered the identified TD themes through coding and the second focused on the relation between some TD themes and patterns.

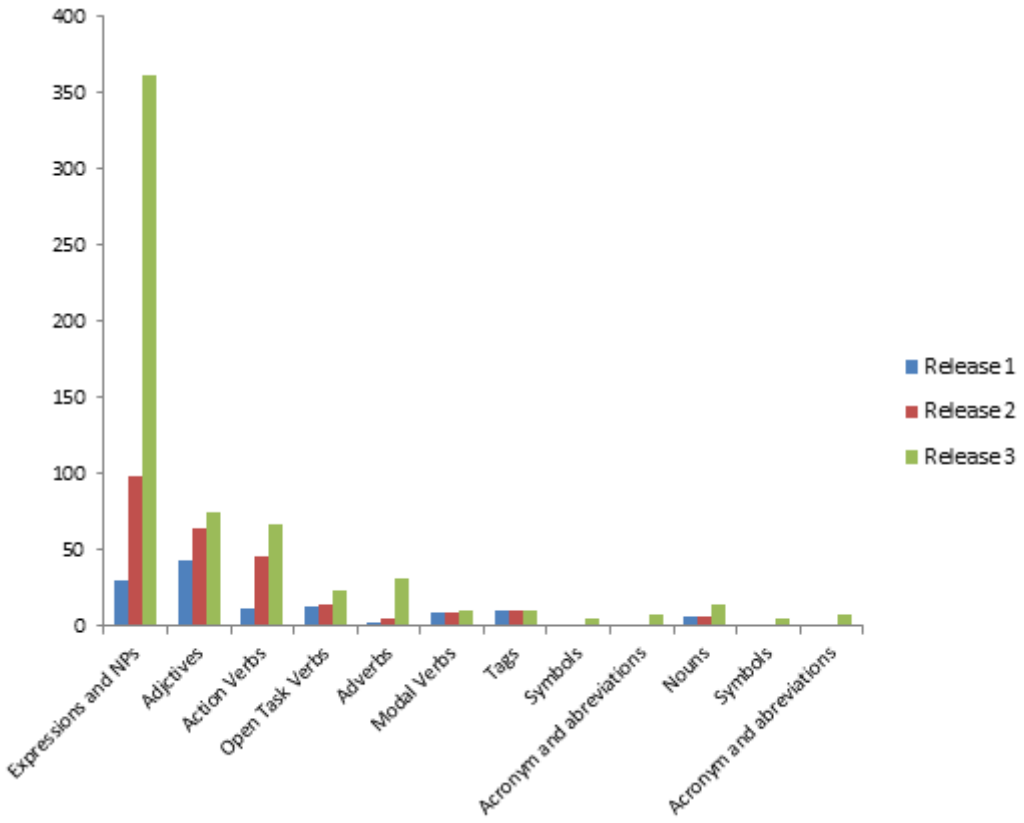


Figure 6.2 Distribution of word classes, tags, expressions, and NPs by releases.

TD themes. In total, we identified 37 different TD themes related to patterns, considering the relationships presented in Section 3.1.3. Table 6.8 presents the set of themes built through the coding method.

Relationship between TD themes and patterns. Besides identifying TD themes, we associated them with a set of patterns. Table 6.8 summarizes the number of patterns associated with each TD theme.

This association can be used to identify specifics TD themes in software projects through identification of patterns into code comments. The complete list is publicly available [29].

Analyzing Table 6.8, it is possible to observe that the theme "Open task" has the highest number of associated patterns (116 patterns), the second most associated pattern is "Low/bad external/internal quality of properties" (65 patterns), followed by "Asking Question" (52 patterns). Patterns related to the first theme can describe tasks needing to be done in the future. Patterns related to the second one can indicate bad quality software artifacts. Whereas the last one expresses doubt and uncertainty about software artifacts. The themes less related to patterns are "Violation of modularity" and "Insufficient code coverage" with only 1 identified pattern.

From the complete list, we can also note that, although many patterns have been

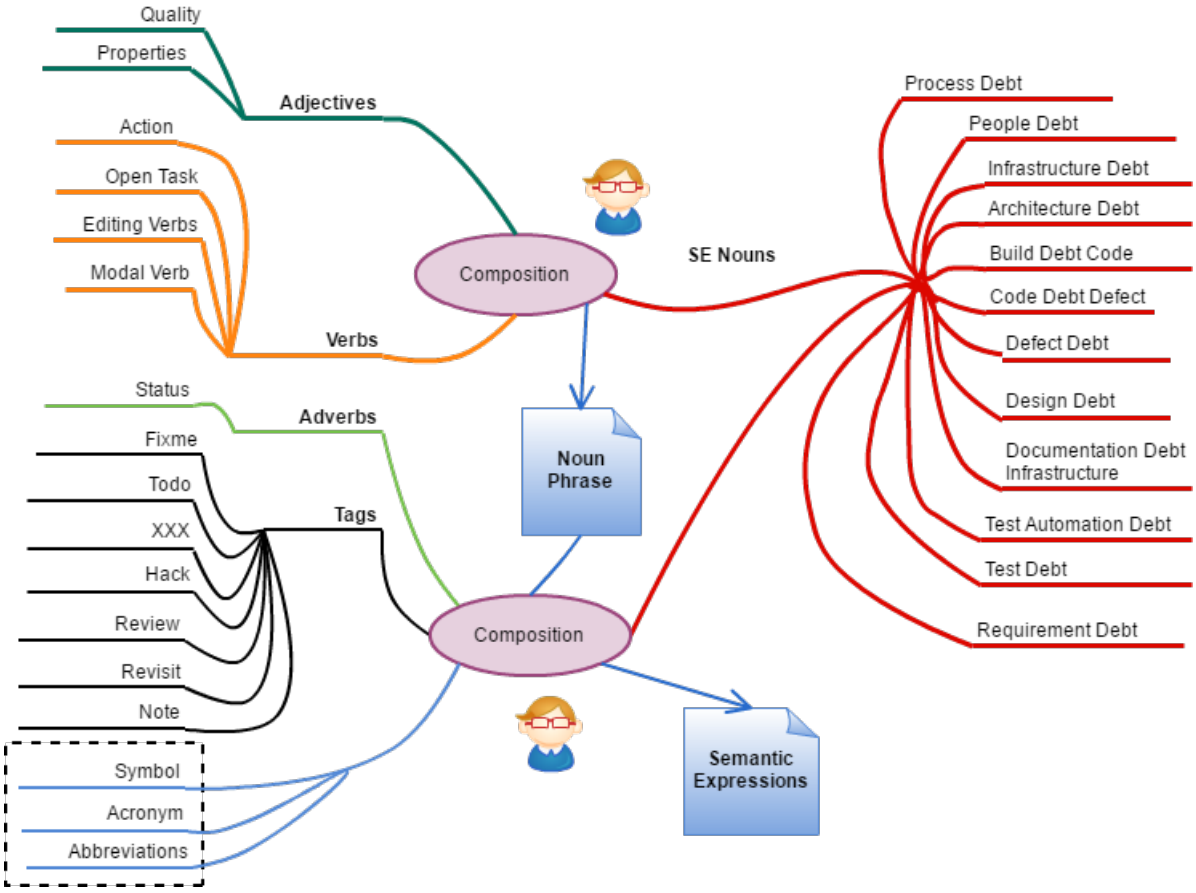


Figure 6.3 New CVM-TD.

related to at least one TD theme, we were not able to associate 130 patterns with any TD theme. In general, these patterns are action verbs or modal verbs.

Relation between TD themes and TD indicators. To address the RQ4, we regarded the relationships between patterns, TD themes and types of TD described in Section 3.1.3

Table 6.9 shows the relationship between TD themes and TD indicators. We observe that indicators were not identified for some themes. In the cases where we were not able to associate themes with indicators, we related directly some themes to one or more types of TD using information provided by each definition of TD type, for example, the theme "tests to do" was not associated with any TD indicator, but it was related to test debt. For other ten themes (e.g. "Task should not be done" and "Open task"), we were able to associate them neither with TD indicators nor with types of TD. In this case, these themes are associated with patterns that can describe a TD item, but cannot identify the type of TD. After this step, we notice that not all TD indicators could be associated with TD themes, for example, "unnecessary code forks" and "referential integrity constraints". See [13] and [6] for more details about these indicators. After relating TD themes to TD indicator, we might associate TD themes with TD type, as can be seen in Figure 6.4.

Table 6.8 Themes related to TD contexts.

TD Themes	Total of Patterns	TD Themes	Total of Patterns
Open task	116	Violation of principles of good design	8
Expressing low/bad external/internal quality of properties	65	Task to be solved	8
Asking Question	52	Can causes exception	7
Status of the work	27	Misunderstanding	7
Inadequate location	18	Dependency	6
Reporting issues or problems	18	Bad coding practices	6
Inadequate solution	17	Deprecated code	6
Difficult to be maintained in the future	15	Need to be safe	6
Not according	15	Temporary solution	5
Task should not be done	14	Performance or, Robustness problems	4
Defects or bugs	13	Human factors	4
Uncertainty about any artifact	12	Unnecessarytime or memory consuming	3
Uncorrected known defects or bugs	11	Redundancy	3
It is not working or does not work very well	9	Code smells	3
Documentation (missing, inadequate, incomplete, or outdated)	9	Deficiencies in testing activities	3
Unnecessary code	8	Not needed	3
Need to implement or it wasn't well implemented	8	Tests to do	2
Duplicate code	8	Violation of modularity	1
		Insufficient code coverage	1

Analysis of occurrences of the patterns and TD themes in studied projects. Table 6.10 presents the occurrences of patterns identified in comments from ArgoUML, jEdit, and Lucene. The number of occurrences is organized by themes and projects. As shown in Table 6.10, we confirm that "Open Task" is the most common theme. It has the highest occurrences of patterns in all studied projects (260 occurrences). As described above, this theme is not associated neither with TD indicators nor with TD types, but

it is important to support the identification of TD because the patterns associated with it denote works that need to be completed or that should be done. For example:

```
Comment #1:
/* TODO: This functionality needs to be moved someplace useful*/
```

The above comment has two patterns describing a task to do, "TODO:" and "need to be moved". It describes the necessity of moving a functionality to a useful place. This comment could also describe a design debt.

We found occurrence of almost all themes in comments from the projects, totaling 691 instances of patterns. This shows that the patterns composing each theme can really be useful to support the TD identification. Besides, the majority of themes is associated with some type of TD and can be used to classify a TD item into one or more types of TD; only 10 themes are not associated with.

During the analysis, we noted that some comments can report more than one type of debt. Let's consider the following comment:

```
Comment #2:
// An alternative implementation of the parsing of pathitems is to
collect // everything at the start, then iterate through it all at
the end.  // The code below does this - it works, but it is currently
not used, // since it is a unnecessarily complicated.  // There are
probably better ways to implement this than using an // ArrayList of
Hashtables (ArgoUML)
```

This is an example of a comment we consider as design and code debt because it has patterns related to themes associated with both types of TD. In the above comment, the developers describe workarounds, code issues, and the necessity for improving the current design and code.

Table 6.9 TD Themes x TD indicators (sorted by TD types).

TD Type	TD Themes	TD Indicators
Architecture debt	Violation of modularity	Violat. of Modularity
	Performance or,Robustness problems	Software architecture issues
Archit/Build debt	Dependencies	Structural depend.
Build debt	Unnecess. time or memory consuming	Build issues
Code debt	Difficult to be maintained in future	-
	Bad coding practices	Code without stand.
	Unnecessary,or not used code	-
	Deprecated code	Code without stand.
	It is not working or does not work very well	-
	Can causes exception	-
	Redundancy	-
Code/ Requirement debt	Need to implement or it wasn't well implemented	-
Requirement	Need to be safe	-
Code debt/ Design debt	Duplicate code	Code without stand.
	Temporary solution or workarounds	-
	Inadequate solution	-
Design debt	Expressing low/bad external/ internal quality of properties	Low external/internal quality
	Violation of principles of good design	Software design issues
	Inadequate location	
	Code smells	Code smells
Defect debt	Defects or bugs	-
	Uncorrected known defects or bugs	Uncorr. known defects
Documentation debt	Documentation (missing, inadequate, incomplete, or outdated)	Lack of documentation and Documentation
	Misunderstanding	
Test debt	Tests to do	-
	Deficiencies in testing activities	Incomplete tests
	Insufficient code coverage	Insuff.code coverage
People debt	Human factors	-
-	Uncertaintyabout any artifact, Asking Question, Open task, Task to be solved, Reporting issues or problems, Necessity of understanding, Task should not be done, Not needed, Not according, Status of the work, To do better (Improvements).	-

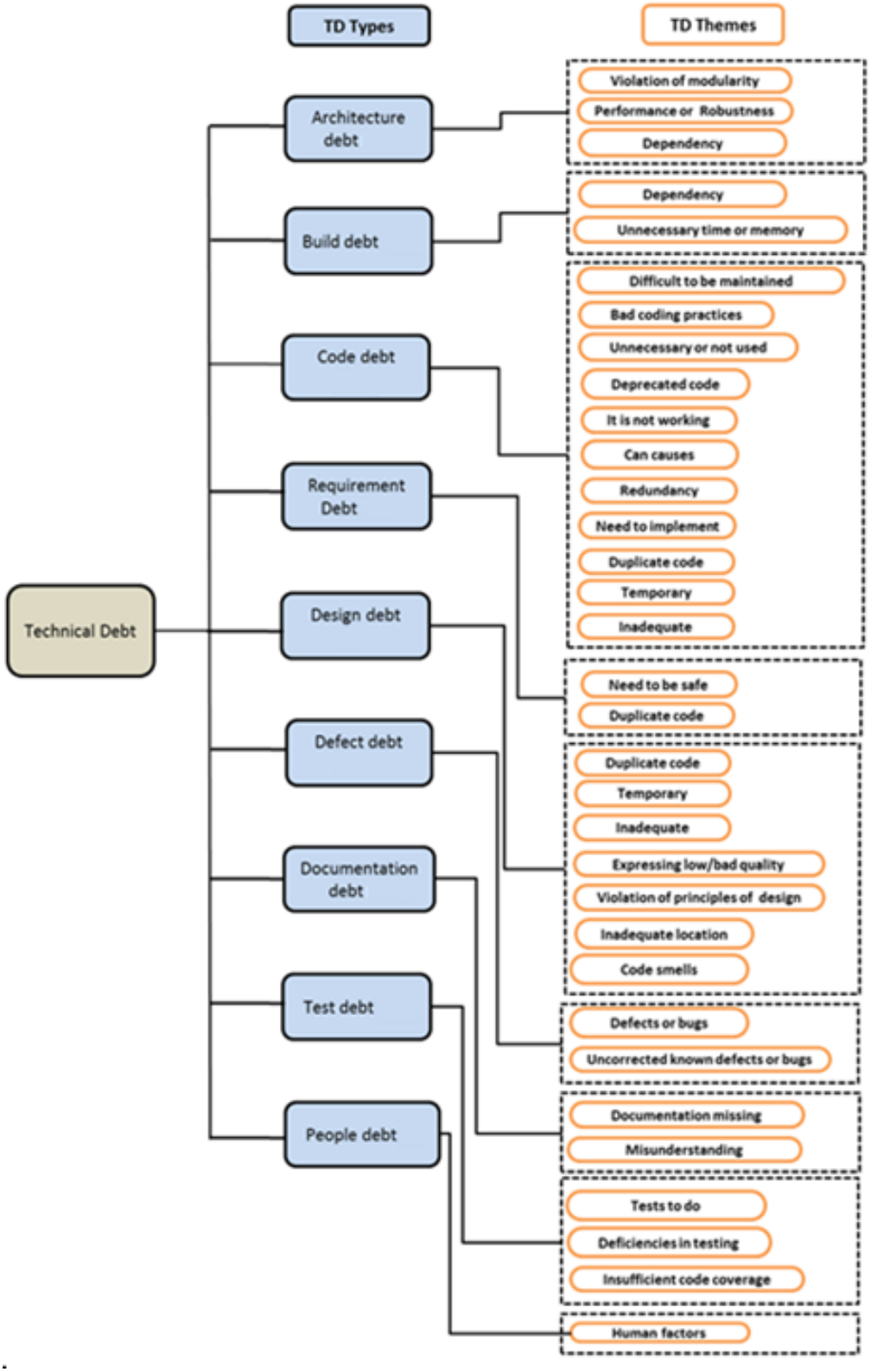


Figure 6.4 TD themes x TD types.

Table 6.10 Total of patterns (comments) by TD themes and OSS projects.

TD Type	TD Themes	Total of occurrences by OSS project(Total: 690)			
		Argo	**jEdit**	**Lucene**	**Total**
Architecture debt	Perform. or Robustnes problems	-	-	1	1
Archit/Build debt	Dependencies	8	-	2	10
Build debt	Unnecess. time or memory consuming	2	-	1	3
Code debt	Difficult to be maintained in future	8	3	2	13
	Bad coding practices	1	-	1	2
	Unnecessary,or not used code	6	-	2	8
	Deprecated code	2	1	-	3
	It is not working or does not work very well	9	12	7	28
	Can causes exception	10	1	-	11
Code/ Requirement debt	Need to implement or it wasn't well implemented	10	6	1	17
Requirement	Need to be safe	4	2	2	8
Code debt/ Design debt	Duplicate code	7	2	1	10
	Temporary solution or workarounds	6	2	-	8
	Inadequate solution	13	2	-	15
Design debt	Expressing low/bad external/ internal quality of properties	7	9	8	24
	Violation of principles of good design	3	1	2	6
	Inadequate location	13	4	4	21
	Code smells	1	2	-	3
Defect debt	Defects or bugs	11	12	4	27
	Uncorrected known defects or bugs	5	4	1	10
Documentation debt	Documentation (missing, inadequate, incomplete, outdated)	4	1	-	5
	Misunderstanding	1	1	1	3
Test debt	Tests to do	2	3	3	8
	Deficiencies in testing activities	1	-	3	4
-	Uncertaintyabout any artifact	13	6	15	34
-	Asking Question	37	3	7	47
-	Open task	139	60	61	260
-	Task to be solved	4	1	2	7
-	Reporting issues or problems	10	1	8	19
-	Task should not be done	9	-	7	16
-	Not needed	3	-	-	3
-	Not according	13	1	7	21
-	Status of the work	7	2	1	10
-	To do better (improvements)	16	3	6	25

We can also observe that some types of TD, such as code and design, already have a fair number of TD themes associate with. On the other hand, not all types of TD proposed in [13] could be associated with themes, such as infrastructure, test automation, service, usability, and versioning debt. We think some types like infrastructure or versioning debt are less probable to appear in code comments but a further study needs to be performed to evaluate this case.

Using TD themes to detect which types of TD exist in the studied software projects. A TD item can be incurred at any moment in the software development life cycle and may be related to several issues, such as bad design, incomplete documentation, and missing tests. These immature artifacts may be seen as a type of debt that may burden software maintenance in the future [13].

Different types of debt can bring different consequences to the software project. Thus, quantifying the TD by its type can help developers understand what they need to consider when deciding if a debt should be paid or not.

In order to evaluate the relationship between TD themes and types of TD and address the RQ5, for each OSS projects (ArgoUML, jEdit, and Lucene), we performed a qualitative analysis on code comments and quantified the number of types of TD that exist using the comment patterns and their related TD themes. Thus, after coding, we applied the TD themes aiming to identify comments related to specific types of TD.

Figure 6.5 and Table 6.10 summarize the number of comments reporting some types of TD in each OSS project. We found eight different types of TD in the three studied projects. From this figure, we highlight that code, design, and defect debt are the most frequent in all projects. We found 129 comments describing code debt in the projects, 77 comments in the ArgoUML, 32 in the jEdit, and 20 in the Lucene. The next most frequent type is design debt with 85 comments, 48 in the ArgoUML, 22 in the jEdit, and 15 in the Lucene, followed by defect debt with 43 comments, 19 in the ArgoUML, 16 jEdit, and 8 in the Lucene. It is important notice that one pattern can be found into one or more comment and one comment can have one or more patterns.

These three types of TD represented 82.37% of the occurrences and the remaining types have low frequency considering that they represented 17.63% of the occurrences. The type less frequent is documentation debt with only 8 identified comments. A possible explanation for this is that developers write about code, design, and defect debt in code comments because these types are more related to the source code. Even though general trend, we identify types of TD that are not directly related to source code, such as documentation and requirement debt.

Part of these results are in line with the findings of Maldonado and Shihab [76] and Silva et al. [77], which found also design and defect debt as the most frequent debts and documentation and build debts are less frequent in code comments. In both studies, the authors did not consider code debt.

Finally, we detail the types of TD that we identified into code comments from studied projects based on [6] and provide some example comments for each type.

Architecture debt: comments that describe this debt refer to the problems encountered in project architecture, for example, violation of modularity, which can affect architectural requirements, such as performance, robustness, among others. We can see

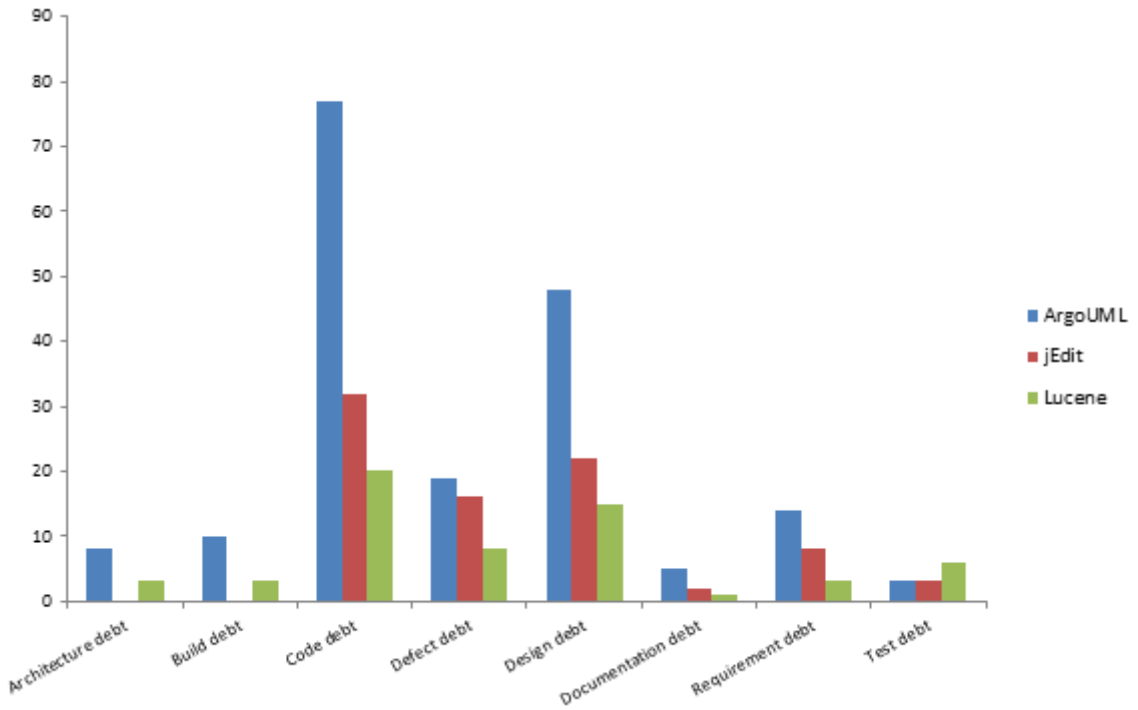


Figure 6.5 Types of TD by studied OSS projects.

this context in below comment:

```
Comment #3:
/* ...  This could possibly hurt performance if the terms dict is
not hot since OSs anticipate sequential file access.  the writer to
re-order the blocks as a 2nd pass.  - Each block encodes the term
suffixes packed sequentially using a separate vInt per term which
is 1) wasteful and 2) slow (must linear scan to find a particular
suffix).  We should instead 1) make random-access array so we can
directly access the Nth suffix and 2) bulk-encode this array using
bulk int[] codecs then at search time we can binary search when we
seek a particular term.  (Lucene)*/
```

Build debt: comments refer to build related issues that make a task harder, and more time/processing consuming unnecessarily. The two comments below relate strategies that caused a very large processing consuming in the build process of the projects.

```
Comment #4:
// NOTE: this test will fail w/ PreFlexRW codec!  (Because // this
test uses full binary term space but PreFlex cannot // handle this
since it requires the terms are UTF8 bytes).  // Also SimpleText
codec will consume very large amounts of // disk (but should run
successfully)...  (Lucene)

Comment #5:
// The following debug line is now the single most memory consuming
// line in the whole of ArgoUML. It allocates approximately 18% of //
all memory allocated ...  (ArgoUML).
```

Code debt: code debt comments refer to the problems found in the source code which can negatively affect the quality of the code making it more difficult to be maintained in the future. This debt can be identified by examining comments considering issues related to bad coding practices (complex code, low-quality code, duplicate code, code outside of standards, deprecated code, etc), as described in the following comments:

```
Comment #6:
/* Note:  This class is messy.  The method and field resolution
need to be rewritten.  Various methods in here catch NoSuchMethod or
NoSuchField exceptions during their searches (jEdit)

Comment #7:
/* TODO: Replace the next deprecated call.  This case is complicated
* by the use of parameters.  All other Figs work differently.  */
(ArgoUML)

Comment #8:
// XXX: getMarkPosition() deprecated!  (jEdit)

Comment #9:
// REVIEW This is horribly inefficient, but it ensures that we //
properly skip over bytes via the TarBuffer...  // (jEdit)

Comment #10:
// Look for a default invoke() handler method in the namespace //
Note:  this code duplicates that in This.java...  should it?  // Call
on 'This' can never be a command (jEdit)
```

In the above comments, the developers describe situations correlated with following themes: "Bad coding practices" (comment #6), "Deprecated code" (comments #7 and #8), "Difficult to be maintained in the future" (comment #7), "Low/bad external/internal quality of properties", "Code is not working or does not work very well" (comment #9), and "Duplicate code" (comment #10). They are examples of what we consider as code debt.

Defect debt: Defect comments refer to known defects, bugs, or failures found in software systems. Defect debts are usually identified by developers in code comments using information reported on bug tracking systems or identified into source code. For example:

```
Comment #11:
// TODO: This class is part of a dependency cycle with ProjectBrowser
and // GenericArgoMenuBar, but should be fixed if project open/close
is moved // out of ProjectBrowser (ArgoUML)

Comment #12:
// This is a workaround to ensure setting the // focus back to the
textArea.  Without this, the // focus gets lost after closing the
popup in // some environments.  It seems to be a bug in // J2SE 1.4
or 5.0.  Probably it relates to the // following one (jEdit)

Comment #13:
/* // NOTE: intentionally fails!  Just trying to debug this //
one input...  public void testDecomposition6() throws Exception...
(Lucene)
```

The above comments mention bugs or fails that need to be fixed – in other words, parts of the code that do not have the expected behavior. In general, developers agree that defects are a problem for the projects and should be fixed but for some reason have to be deferred to a later time.

Design debt: In design debt comments, developers describe technical shortcuts and design issues. These comments can indicate design debt by identifying the use of practices which violated the principles of good object-oriented design, such as very large, complex class or method, tightly coupled classes, code smells, workarounds or a temporary solution. In accordance with next comments, we can notice themes related to design debt.

```
Comment #14:
/* Thanks to Slava Pestov (of jEdit fame) for import caching
enhancements.  Note:  This class has gotten too big.  It should be
broken down a bit.  */ (jEdit).

Comment #15:
Note:  this implementation is temporary.  We currently keep a flat
namespace of the base name of classes.  i.e.  BeanShell cannot be in
the process of defining two classes in different packages with the
same base name (jEdit).

Comment #16:
// This is somewhat inconsistent with the design of the constructor
// that receives the root object by argument.  If this is okay // then
there may be no need for a constructor with that argument (ArgoUML).

Comment #17:
...//TODO: This has a bad smell.  I don't think we should be using
Modes here...  (ArgoUML)
```

Documentation debt: Documentation debt comments express problems found in software project documentation. In the comments below, developers recognize outdated code documentation, insistent documentation, and the need to document the code. Here, developers show their worries about missing or inadequate documentation.

```
Comment #18:
* TODO: needs documenting, why synchronized?  */ (ArgoUML)

Comment #19:
/*TODO: As currently coded, this actually returns all
BehavioralFeatures which are owned by Classifiers contained in the
given namespace, which is slightly different than what's documented.
It will not include any * BehavioralFeatures which are part of the
Namespace, but which don't have an owner.  */ (ArgoUML)

Comment #20:
// undocumented hack to allow browser actions to work.  // XXX -
clean up in 4.3 (jEdit).
```

Requirement debt: Comments describing requirement refer to lack of synchronism between optimal requirements specification and what is currently implemented. The comments below express some examples of this type of debt, such as lack of implementation (comment #1), requirements that are only partially implemented (comment #2), requirements that are implemented but not for all cases (comment #3), and requirements that are implemented but in a way that doesn't fully satisfy all the non-functional requirements (e.g. security, performance, etc).

```
Comment #21:
/* TODO: Add implementation.  */ (ArgoUML).

Comment #22:
/* TODO: The copy function is not yet completely implemented - so we
will * have some exceptions here and there.*/ (AlgoUML).

Comment #23:
// Temporary hack to support inner classes // If the obj is a
non-static inner class then import the context...  // This is not
a sufficient emulation of inner classes.  // Replace this later...
```

Test debt: Test comments refer to shortcuts taken in testing. Some examples are lack of tests and deficiencies in testing activities, as can be seen in example comments below.

```
Comment #24:
// TODO: Do we really have to test for null here?  // classifier
role does not exists, so create a new one Object newClassifierRole =
Model.getCollaborationsFactory ...  (ArgoUML)

Comment #25:
/* But it is not reliable as a unit test since it is
timing-dependent public void testManyRepeatedTerms() throws Exception
...  */ (jEdit)
```

6.8 SUMMARY OF FINDINGS AND INSIGHTS

The results show that many patterns identified in *FindTD II* were indicated as decisive to select comments describing a TD context. We analyzed the 25 most selected patterns by participants, and only six of them were not mentioned as decisive or very decisive to identify a TD item. Regarding all patterns established in the *FindTD II*, we found that 52.72% of patterns were identified as "decisive" or "very decisive", 28.94% of patterns were identified as "not much decisive", 13.47% were indicated as a pattern that "can be decisive", and 4.58% were identified as "not decisive" to identify a TD idem. This result shows that over half of the patterns defined in *FindTD II* are considered decisive or very decisive to identify TD.

We also identify that 55.93% of the composed patterns were scored as "very decisive" or "decisive", and only 35.19% of the isolated patterns were chosen as "very decisive" to identify a TD item. This evidence shows that composed patterns seem to be more contextualized and decisive than isolated patterns to identify TD. To carry out a deep analysis, we separately analyzed the scores of the patterns and after we analyzed the scores when the patterns appeared combined with other patterns. The results show that some patterns only make sense when they are combined with other ones. We also identify that tags are relevant to compose decisive patterns aiming to report a TD context.

Concerning to comments that were chosen as a good indicator of TD in *FindTD II*, the results show that comments composed by decisive patterns were indicated as a good TD indicator. It means that decisive patterns may help to filter valuable comments for reporting TD and help developers to comprehend a TD context described into code comments. From this evidence, we can set thresholds to be used in *eXcomment* to select comments by considering the level of contextualized information to detect those that may point out to a TD.

To identify TD themes, we used coding technique and focused on the relationship between some TD themes and patterns. We identified 37 different TD themes related to patterns which describe a TD context. Through these themes, we could identify comments related to them and different types of TD. We found occurrences of almost all themes in comments from the projects. TD themes can be used to detect TD identifying specific scopes associated with each type of TD. Also, we mapped these TD themes and TD indicators proposed by [13].

We used TD themes to detect which types of TD exists in the software projects explored in three performed experiments thought code comment analysis. We found 8 different types of TD in the three studied projects. We highlight that code, design, and defect debt are the most frequent in all projects. However, even though general trend, we identify types of TD that are not directly related to source code, such as documentation and requirement debt. This evidence shows that code comments can be used to identify code-based types of debt and debts that were undetectable using only code metrics.

In the next study, *FindTD IV*, we evaluate our complete approach to identify and classify TD items through code comments analysis.

6.9 THREATS TO VALIDITY

We followed the checklist provided by [94] to discuss the relevant threats to this study.

6.9.1 Construct Validity.

The first limitation is regarding to the technique used to perform the qualitative analysis. We adopted the coding techniques in order to conduct an analysis of the patterns related to different types of TD. However, we have to consider that other qualitative data analysis strategies would offer different perspectives of the knowledge on the patterns belonging to the vocabulary. To mitigate this threat, an expert research will supervise the results in order to verify the consistency of our data analysis and the coding.

Another threat involves the definition of the proposed vocabulary. It is possible that the set of patterns and combinations used by our model and vocabulary are simply too many to be studied. An alternative would be to limit the studies to specific contexts and software domains.

6.9.2 Internal Validity.

The first internal threat we have to consider is subject selection, since we have chosen all participants through a convenience sample. We randomly split the participants in different treatment groups in order to minimize this threat. Another threat is that participants might be affected negatively by boredom and tiredness. In order to mitigate this threat, we performed a pilot study to calibrate the time and amount of comments patterns to be analyzed.

In addition, since we had to infer TD themes using a qualitative analysis, one might argue that the categories are subjective. We followed a systematic approach and triangulation to mitigate this threat.

6.9.3 External Validity.

Even though graduate student analyzed data from real software, we cannot generalize the findings and their applicability to industrial practices. We chose a large and mature open source software projects to try mitigating this thread. All participants of the experiment have some professional experience in the software development process. It is an important aspect in mitigating the threat.

The last limitation covers the effort to carry out all studies because of the difficulty of performing experiments in this area, for example, the threat relates to the generalization of the findings and their applicability to industrial practices. This threat is always present in experiments with students as participants.

6.9.4 Conclusion Validity.

Conclusion validity threats are mainly due to avoid the violation of assumptions. We limited our data analysis at considering simple statistic methods and qualitative analysis. The findings considered the data analysis of four participants in the experiment in three

OSS projects. To avoid the violation of assumptions, we considered patterns identified in the FindTD II [25] by different participants for data analysis.

Chapter 7

This chapter presents the last study of the family of experiments. We evaluated the process to identify and classify TD items through code comment analysis automatically. We applied qualitative and quantitative data analysis. The results allowed us to investigate the contribution of the vocabulary to support the TD identification process. We concluded that code comments provide a perspective which permits to consider the developer's point of view to complement the quantitative analysis to identify TD items.

FINDTD IV

7.1 GOAL OF STUDIES AND RESEARCH QUESTIONS (RQ)

This study aims to **evaluate eXcomment with the purpose** of analyzing its capacity to score and classify TD items, **with respect to** the accuracy **from the point of view** of the researcher **in the context of** graduate students and software engineers analyzing comments classified as TD items.

The experiment will provide us new evidence on how code comments can be explored in order to identify and classify automatically TD items. We noticed that this evidence will support the *eXcomment* evolving, making it more contextualized and accurate to identify and classify TD items.

The main research question, which guides our study, aims to investigate if it is possible to automatically identify TD items through code comments using a contextualized vocabulary. To achieve the overall goal, we defined four secondary research questions:

RQ1*: Does our contextualized vocabulary help researchers in the selection of candidate comments that point to technical debt items?*

With this question, we intend to broaden the investigation (performed in *FindTD II*) on the comments selected by the contextualized vocabulary, evaluating whether they really describe a TD situation. This will also allow us to investigate the contribution of the vocabulary to support the TD identification process.

RQ2*. Do the eXcomment scores represent how much a comment describes a TD situation?*

eXcomment defines final scores that represent how much a comment describes a TD situation. This question aims to investigate whether the calculation of final scores really represents the participants' opinions considering how much a comment can point out

to a TD item. In order to analyze this variable, we consider the scores calculated by *eXcomment* and the values scored by the participants.

***RQ3**. How precise is eXcomment to identify TD themes?*

This research question aims at empirically investigating how precise our approach is to classify the comments in different TD themes. TD themes are a category of patterns which are related to a specific TD situation, for example the theme Open Task is related to tasks that need to be done. In particular, we want to know the most identified themes by the participants in conformity with *eXcomment*.

***RQ4**. How can TD themes support the TD items classification?*

In this research question, we firstly investigate how TD themes identified in the *FindTD III* support the classification of different types of TD. Then, we investigate which themes are useful to support the identification of TD types in order to better understand the relation between themes and types.

***RQ5**. How precise is eXcomment to identify types of TD?*

We have discussed how our approach can be efficient to identify TD items through code comment analysis using a contextualized vocabulary. However, previous experiments did not discriminate among different types of TD. Then, this question aims to discuss how precise our approach is to classify TD items in different types using code comment analysis. Answering this question is important because it helps us identify the limitations by using a contextualized vocabulary to identify and classify TD items.

7.2 APPROACH

The main goal of this study is to evaluate the process of automatically identifying TD items through code comment analysis. The output of this process is a list of comments that report TD instances in both analyzed projects. Figure 7.1 depicts the main steps behind the process of this experiment.

To do that, first, we selected projects and participants for the controlled experiment. We chose ArgoUML and JFreChart OSS as projects to be studied and 22 participants to analyze the comments. As the second step, *eXcomment* mines the entire source code from the selected projects to extract their comments in chronological order. Next, the set of retrieved comments is analyzed by using the fourth release of the contextualized vocabulary to select those comments reporting TD items. In the classification step, comments were classified by *eXcomment* considering their final scores and types of TD. After that, we asked participants to analyze the set of comments classified by the *eXcomment* in order to point scores for each of them and choose their type of TD. Finally, we performed a quantitative analysis considering the accuracy to measure the agreement between our approach and the participants' answers. Next, we performed a qualitative analysis to deeply investigate the content on the comments to complete the first analysis, identify false-positives and examine some specific cases.

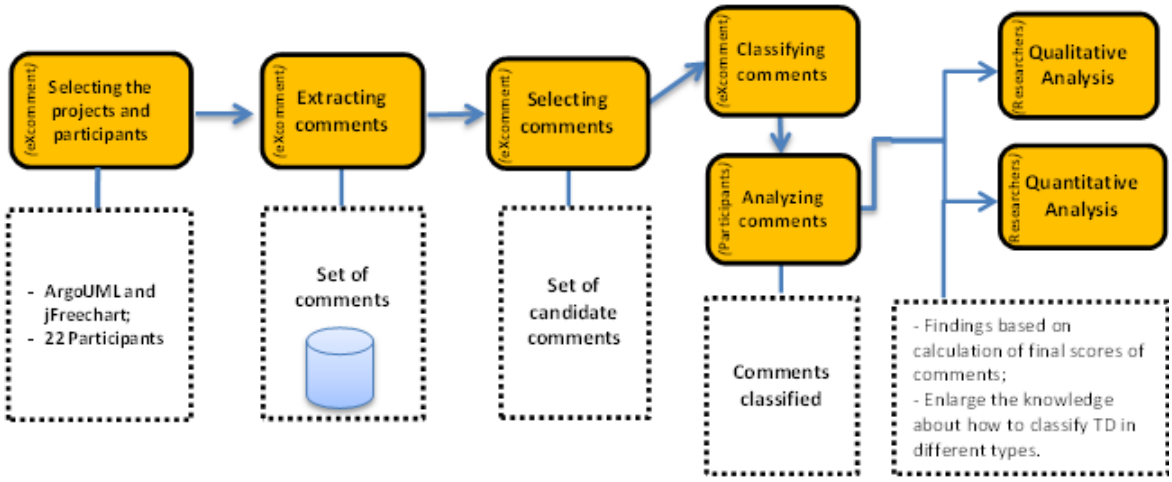


Figure 7.1 Process of FindTD IV.

7.3 PARTICIPANTS

The participants were selected using convenience sampling [20]. Our sample consists of 15 master degree students from the Federal University of Bahia and 7 software engineers from the Fraunhofer Project Center at UFBA.

Each participant filled out a characterization form to provide information on their experience regarding the software development process. We used this information to classify the participants' profile and their experiences in the software development process. The results showed that most of the master degree students had high experience level. Only two students had less than 5 years of experience in programming, but all had some experience on software projects. Regarding software engineers, most of them had more than 10 years of experience in software development. Regarding English reading, all participants answered that they were able to read and understand short English texts (e.g. source code comments).

The participants were randomly divided into two groups. We considered the classification proposed by [86] and presented in Table 6.1 in Section 6.2 to classify the experience levels (high, medium and low) of the participants. We had 21 participants with high experience level, and one with low experience. Thus, the bias from randomization is limited because almost all participants had high-level experience.

The design was done in such a manner that each group analyzed a software project. Each comment from Argouml and JFreeChart was analyzed by 11 participants. We adopted this plan because it allows that a comment should be analyzed by many participants in order to avoid bias due to personal interpretation. Table 7.1 shows the participants' distribution in this experiment.

Table 7.1 Distribution of the participants among groups in FindTD IV.

Group	Participants by experience level		
	High	**Medium**	**Low**
G1_ArgoUML	11	-	-
G2_JFreeChart	10	-	1
Total (22)	**21**	**-**	**1**

7.4 INSTRUMENTATION

7.4.1 Forms

We used slides for the training, three forms, and one web application form to perform the experiment:

Consent form: the participants authorized their participation in the experiment and indicated to know the nature of the procedures which they had to follow.

Characterization form: contains questions to gather information about professional experiences, English reading skills, and the participants' specific technical knowledge. We used a web form (in Portuguese) available at https://goo.gl/9fvZzE to characterize the participants.

Data collecting web application: we used a web application to gather data in the study, as we can see in Figure 7.2. The application form contains the comments to be analyzed and allows to control the participants' access, their answers, and time. During the experiment, participants analyzed the comments individually (1) and they were asked to score on an ordinal scale from one to ten the level with which each comment describes a TD situation (where one means that a comment does not describe a TD situation and ten means that a comment describes clearly a TD item) (2). Following, the participants classified the TD theme for each comment analyzed (a comment can describe more than one TD theme). The *eXcomment* suggests the themes regarding the comment and the participants answer whether he/she agree or not with the classification carried out. A theme references to a situation that can point to a TD item, for example the theme Open Task is related to tasks that need to be done (3). After that, the participants chose the type of TD related to each comment analyzed. The application presents a list with ten different types and they chose one, more than one or no type of TD (4). In the field "Note" (5), participants can write a note about the comment. Finally, participants may access the source code to complement the analysis of each comment in order to help them make the decision on scores and the TD types classification. The participants' answers are stored in a relational database to facilitate the experimental data processing.

Feedback form: in this form, the participants may write their impression and difficulties on the experiment. We also asked the participants to classify the training and the level of difficulty performing the study tasks. We used these data to complete our qualitative analysis. We used a web form (in Portuguese) available at https://goo.gl/T4VjHL to collect feedback information.

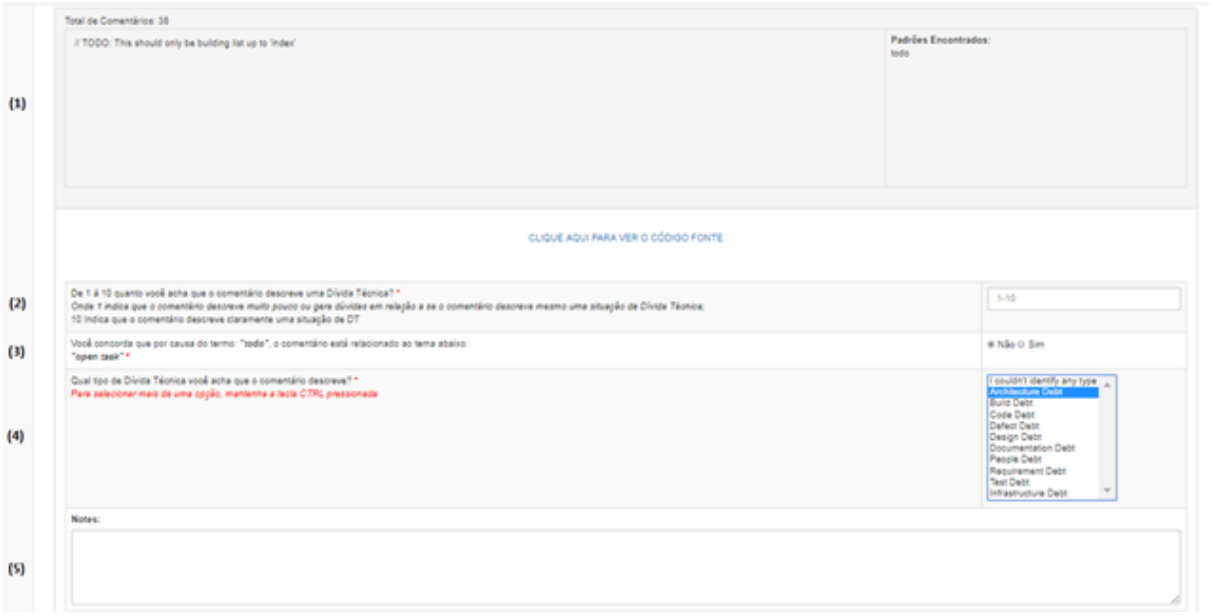


Figure 7.2 Data collect web application - FindTD IV.

7.4.2 Software Artifact and Candidate Comments.

We used comments extracted and filtered from Argouml and JFreeChart to perform our study. Argouml[1] is an UML modeling tool that includes support for all standard UML 1.4 diagrams, whereas JFreeChart[2] is a chart library for the Java language that makes it easy for developers to display professional quality charts in their applications. For choosing these projects, we considered the following criteria: be long-lived (more than 10 years), have a satisfactory number of comments (more than 10,000 useful comments), belong to different application domains, and vary in size.

To be able to extract the candidate comments from the software that may indicate TD items, we used *eXcomment*. We were only interested in comments that have been intentionally written by developers, as discussed in Chapter 3. Once the comments were extracted, we filtered the comments by using patterns that belong to the vocabulary. We will call these comments 'candidate comments' (comments after filtering) in this thesis.

Table 7.2 provides details about each of the projects used in our study. The first column presents the software project, followed by the release used, the age of each project, the number of files, the total source lines of code, and the total extracted comments.

Table 7.3 summarizes the percentage of comments by score levels and by TD types in each of the projects. We can notice that in Argouml, 10.89% of comments had scores higher than 7.0, 54.35% of comments had scores between 7.0 and 3.0, and 34.96% of comments had scores lower than 3.0. Regarding the types of TD, 52.20% of comments were classified as code debt, for example. In JFreeChart, 14.03% of comments had scores higher than 7.0, 66,85% of comments had scores between 7.0 and 3.0, and 19.13% of

[1]Argouml URL: http://argouml.tigris.org/
[2]JFreeChart URL: http://www.jfree.org/JFreeChart/

Table 7.2 OSS projects Metadata of FindTD IV.

Software	Release	Age (years)	# of java files	# of code lines	# of comments
ArgoUML	0.34	14	2,609	176,839	27,298
JFreeChart	1.0.19	17	1,065	132,296	21,551

comments had scores lower than 3.0. Regarding the types of TD, 70.15% of comments were also classified as code debt.

In order to select the number of comments to be analyzed in this study, we used stratified sampling techniques to select subgroups within the population [20]. We selected 65 comments from ArgoUML and 65 comments from JFreeChart, considering the distribution presented in Table 7.3. For example, in Argouml, 7 out of 65 comments (almost 10.89%) were randomly selected considering the distribution of the comments that have scores higher than 7.0 Furthermore, we considered the distribution of each type of TD. For example, in Argouml, 31 out of these 65 comments were randomly selected considering the comments that were classified as code debt (almost 52.20%). We followed the percentages of the each TD type identified in this study and presented in Table 7.3 to classify the rest of the comments. This strategy allows that the 65 comments represent the output of the *eXcomment* for each of the projects.

Table 7.3 Proportion of the number of comments by score levels and TD types classified by *eXcomment*.

Software projects	% of comments having scores higher than 7.0	% of comments having scores between 7.0 and 3.0	% of comments having scores lower than 3.0
ArgoUML	10.89%	54.35%	34.96%
JFreeChart	14.03%	66.85%	19.13%
Percentage of comments classified in each of the types			
TD types		**Argouml**	**JFreeChart**
Code debt		52.20%	70.15%
Design debt		14.55%	3.83%
Defect debt		11.88%	18.62%
Requirement debt		9.08%	6.38%
Architecture Debt		5.61%	0.51%
Build debt		5.21%	0.51%
Documentation debt		0.67%	0.00%
Test debt		0.53%	0.00%
Infrastructure debt		0.27%	0.00%

This set of comments was listed in the data collecting web application in the same

order in which they are in the code.

7.5 ANALYSIS PROCEDURE

We used qualitative and quantitative analysis to investigate the RQs, considering the descriptive statistics, the contents of the comments and the participants' notes. We conducted a qualitative study to deeply investigate the content on the comments. The main goal of this analysis is to investigate whether the comments selected by *eXcomment* really describe a TD situation and to understand the divergences between the participants' answers and the *eXcomment*. Another goal is to investigate if complex comments have an impact on the TD item identification. Through this analysis, it is possible to achieve aspects beyond the quantitative analysis. This perspective supports the investigation of all RQs.

RQ1. *Does our contextualized vocabulary help researchers select candidate comments that point to TD items?*

Qualitative analysis on the contents of the comments. We performed a qualitative analysis to deeply investigate the contents of the comments. Two researchers have carefully read all comments selected by *eXcomment* and they pointed out whether the comments describes a TD item or do not (false positive), considering their point of view. After that, we performed a consensus process for each comment that had a divergent opinion.

We consider a comment as false positive when: i) the comment is selected by *eXcomment*, but it describes the behavior of a method, class, variable, or code; ii) the comment is selected because of a pattern that belongs to a software vocabulary (e.g. clone is a software feature). Consequently, the comment does not describe a TD situation; and iii) although our vocabulary makes *eXcomment* classifying a comment as a TD item, it simply describes a note or instruction about the code or software.

With this in mind, we analyzed patterns which impacted on the comments detection that do not describe a TD item. We consider that a pattern has a bad impact when it is responsible for the selection of a false positive comment. For example, the comment #18 was selected as candidate comment by *eXcomment* because of the pattern "will be", but it was also considered as a false positive comment. Thus, we consider that this pattern had a bad impact on the selection of this comment.

```
Comment #18 (JFreeChart) - Class:CombinedCategoryPlot.java
* Returns the bounds of the data values that will be plotted against *
the specified axis.  *
```

Besides the analysis of false positive comments, researchers took notes about the identification of TD items through comment analysis, which we will present in this analysis.

RQ2. *Do the eXcomment scores represent how much a comment describes a TD situation?*

To answer this question, we analyzed the relationship between the participants' scores and the scores calculated by the *eXcomment*. We considered the final participants' score medians as participants' score. The scores represent the importance level in which a

comment describes a TD situation.

To do this, we used two approaches. First, we measured the correlation coefficient among scores pointed out by the participants and the scores calculated by *eXcomment* in Argouml and JFreeChart, independently. The goal of this topic is to analyze whether the participants' scores and scores calculated by *eXcomment* have a positive and high correlation. Second, in order to complement the analysis of the correlation, we analyzed how much the participant scores are different from scores calculated by the *eXcomment*. This allows us to identify and to investigate some comments with high differences.

Correlation of the final scores. We adopted two approaches in order to examine the correlation between participants and *eXcomment* scores. First, we carried out a statistical correlation test [97]. Second, we recalculated new scores using the new vocabulary after the qualitative analysis (excluding the false positive comments). Then, we performed a new comparison between scores indicated by the participants and the new scores.

We adopted the Spearman coefficient as the statistical correlation test. The Spearman coefficient indicates the direction of the association between two variables (e.g. participants scores and *eXcomment* scores). If participant scores tend to increase when *eXcomment* score increases, the Spearman coefficient is positive. When Spearman coefficient is 1, it means that both variables have a perfect correlation. On the other hand, if participant scores tend to decrease when *eXcomment* score increases, the Spearman coefficient is negative [91].

Differences between participants and eXcomment scores values. We considered the difference between the median of the scores indicated by the participants and score calculated by *eXcomment* for each comment. If the difference is zero, it means that the participants' scores are equal to the *eXcomment*'s score. The smaller the difference, the more similar the scores are. For instance, the median of participants' is 6.30 and the final score calculated by *eXcomment* is 6.20 in a comment. Thus, the difference is 0.10, showing that the scores are very similar.

***RQ3**. How precise is eXcomment to identify TD themes?*

In order to investigate which TD themes are precisely identified by *eXcomment*, we calculated the precision values for each comment analyzed in this experiment concerning to the participants. Precision indicates how many TD themes classified by *eXcomment* were also classified by the participants. Precision values are calculated following the Equation 7.1, considering the agreement between *eXcomment* and the participants in the TD themes classification. The numerator in Equation 7.1 represents the judgment of the participants in concordance with *eXcomment*; the denominator represents the number of analysis by the participants in each comment (11 for Argouml and 10 for JFreeChart).

Precision measures how *eXcomment* is working.

$$precision = \frac{(\#agreementbetweeneXcommentandparticipants)}{\#ofanalysisbycomments} \tag{7.1}$$

In particular, we investigated the most common themes by the participants in conformity with *eXcomment*. It is important to highlight that the themes are used to classify the types of TD by *eXcomment* and the participants.

***RQ4**. How can TD themes support the classification of TD items?*

To answer this question, we consider the TD themes classification and the classification of TD types in order to analyze the relationship between themes and TD types. To do that, we explored the precision values, considering the participants' answers and the *eXcomment* classification. The main goal of this analysis is to evaluate whether and how this relation supports the classification of types of TD. Then, we identified the themes that are useful and the themes that are not useful to support the identification of the types. After, we focused on the analysis of the cases where the themes were not considered useful in order to identify the limitation of the relationship between themes and TD types.

RQ5. *How precise is eXcomment to identify the types of TD?*

In order to investigate which types of TD are precisely identified by *eXcomment*, we calculated the precision values for each comment regarding the participants. We did not calculate the recall because we did not consider the false negative comments in this experiment. This analysis is similar to one presented in RQ3. To do this, we used two approaches. First, we measured the precision of the types of TD identified by *eXcomment*. Second, we calculated the precision of comments without types identified automatically by our approach. The main goal of the second analysis is to identify the *eXcomment* limitation to classify some types of TD.

Types identified automatically by eXcomment. In this point, we calculated the precision values regarding the participants for each comment that *eXcomment* could identify at least one type of TD. That is, we analyzed only the comments that *eXcomment* identified some type of TD.

Comments without types identified automatically by our approach. In this point of view, we focused on the cases precision values where the *eXcomment* did not classify any type of TD but the participants found some clues related to the description of some types.

7.6 PILOT STUDY

Before the experiment, we carried out a pilot study with two TD expert researchers to mitigate the risk of providing a biased experiment. The pilot study took 2 hours and was carried out in a remote environment. We performed the training at the first hour, and next the participants performed the experimental task described in the next section. A participant analyzed 100 comments from Argouml and another analyzed 100 comments from JFreeChart.

The pilot study was used to better understand the procedure of the study. It helped us to evaluate the use of the data collected through web application form, the necessary time to accomplish the tasks and, mainly, the number of comments used in this study. Thus, the pilot study was useful to calibrate the time and number of comments to be analyzed in each of the projects. We reduced the number from 100 to 65 comments in each analyzed software.

7.7 OPERATION

The experiment was performed in a classroom at the Federal University of Bahia and at the Fraunhofer Project Center at UFBA, following the same procedure. The training and experiment itself were performed at the same day. For training purposes, we performed a tutorial in the first part of the experiment. The tutorial covered the TD concepts, the different types of TD, and how to perform a qualitative analysis on the code comments using patterns to identify and classify TD items. This training took 1 hour. The participants filled in the consent and characterization form after the training.

After that, there was a break. Next, each participant analyzed the set of candidates comments, extracted from ArgoUML and JFreeChart, in the same room where the training was provided.

The order of the participants was assigned randomly to each set of comments so that each participant analyzed 65 comments and each comment was analyzed by 11 participants. Finally, participants were asked not to discuss their answers among themselves. When they finished, they filled in the feedback questionnaire. The experiment training and execution took a total of three hours.

7.7.1 Deviations from the Plan.

We eliminated one participant of our study (from G2_JFreeChart) because he did not complete all the experimental sessions and we needed all the information to a complete analysis (characterization, data collection, and feedback). Table 7.4 presents the participants' final distribution.

Table 7.4 Final distribution of the participants among groups in FindTD IV.

Group	Participants by experience level		
	High	Medium	Low
G1_Argoum	11	-	-
G2_JFreeChart	9	-	1
Total (21)	**20**	**-**	**1**

7.8 ANALYSIS OF FINDTD IV

In this subsection, we describe the results of the experiment.

7.8.1 Does our contextualized vocabulary help researchers select candidate comments that point to technical debt items (RQ1)?

To evaluate this question, we designed and conducted a quantitative and qualitative analysis to investigate the comments selected by the *eXcomment*, the patterns used to select them as comments that can describe a TD situation, and their contents. This question deepens the discussion performed in FindTD II (discussed in Chapter 5) to a more detailed level.

Comments and patterns selected by *eXcomment*

Table 7.5 shows the total of useful comments and the number of comments after applying our vocabulary. The *eXcomment* extracted 13,114 useful comments from Argouml and 11,578 from JFreeChart. In spite of a comment to be considered useful, it does not mean that it describes a TD item. In fact, a comment is selected by *eXcomment* as comments that can describe a TD situation when it has at least one pattern or one heuristic from the vocabulary. Thus, the *eXcomment* selected 3,097 comments from Argouml and 1,730 comments from JFreeChart as TD comments.

Table 7.5 Extracted comments by project.

Software	# of comment	# of useful comments	# of TD comments
ArgoUML	27,298	13,114	3,097
JFreeChart	21,551	11,578	1,730

Table 7.6 summarizes the 25 most identified patterns in both projects. In Argouml, the most found pattern is the tag "*TODO*" (1,073 occurrences). Next , we present a sample of comments where we identified "*TODO*". We highlight that the comments describe tasks that are still not performed but that can be important for the project. They are evidence that developers write clues about open task in comments.

```
Example Comments (Argouml)
// TODO: Review all callers to make sure that they localize the title"
// TODO: Add an "open most recent project" command so that command
state can be decoupled from user settings?"
// TODO: Allow other configuration handlers."
```

In JFreeChart, the most found pattern is the adjective "*Deprecated*" (280 occurrences). This pattern indicates that an artifact will not be maintained and it should not be more used because it may cause problems in the future. Following, we present a sample of comments where we identified "*Deprecated*".

Furthermore, the patterns can be related among themselves resulting in other composed patterns or in heuristics. Table 7.7 summarizes the occurrences of each heuristics found in both projects. We note that the heuristic "*Open Task Verb + Action Verb*" was the most identified in both projects. This result is in line with results found in *FindTD I* and *FindTD II*, which show that Open Task Verbs such as "*need to*", "*to do*", "*should be*", "*must be*", and "*have to*" are frequently used to describe an open task, whereas Action Verbs such as "*add*", "*fix*", "*change*", and "*remove*" are frequently used to denote activities in software engineering. Thus, this heuristic shows works that need to be completed or that should be done. The second most identified heuristic is "*Modal Verb + Action Verb*". It indicates necessities or possibilities and it also describes open tasks performed or not by developers.

We also note that there is a significant difference between the occurrences of the heuristic "*Tag + Action Verb*" in Argouml and JFreeChart. The reason for this can be related to the fact that tags were more used in Argouml than JFreeChart.

Table 7.6 the 25 most identified patterns in both projects.

	ArgoUML		JFreeChart	
	Pattern	**# of occurences**	**Pattern**	**# of occurences**
1.	Todo:	1073	Deprecated	280
2.	should be	325	will be	269
3.	can be	223	can be	211
4.	deprecated	214	Required	196
5.	will be	191	should be	175
6.	need to	168	Necessary	41
7.	Issue	148	Bug	73
8.	required	136	no longer be used	65
9.	should not be	108	need to	65
10.	Note	98	Note	47
11.	Problem	98	may be	46
12.	dependency	90	Necessary	41
13.	Error	89	temporary	35
14.	must be	86	must be	35
15.	may be	64	Todo	31
16.	Fail	54	Fixme	19
17.	haveto	49	Error	27
18.	in the future	44	need to clone	27
19.	would be	41	should not	26
20.	Later	40	Redundant	25
21.	Warning	38	should not be	21
22.	need to be	36	later	21
23.	Todo: why	34	Duplicate	20
24.	Temporary	32	not implemented	20
25.	Duplicate	14	in the future	17

After identifying patterns and heuristics, *eXcomment* classifies the comments in different themes and types of TD. Analyzing the relationship between patterns and themes, we identified 32 out of 37 TD themes in Argouml and 19 in JFreeChart. Table 7.8 presents the types of TD identified in both projects. The column "# of comments" shows the number of comments classified in each type. The most classified types were code, defect and design debt in both projects. This result is in line with the analysis of *FindTD III*.

We notice that *eXcomment* makes it possible to extract a list of useful comments and the patterns of our vocabulary make it possible to identify TD comments to support the

Table 7.7 Occurrences of each heuristics found in both projects.

	ArgoUML	**JFreeChart**
Open Task Verb + Action Verb	379	224
Modal Verb + Action Verb	248	131
Tag + Action Verb	81	9
Adjective + Noun	59	25
Noun + is/are + Adjective	9	9

TD items identification. However, in spite of the results show evidence that the model and vocabulary can filter TD comments describing a TD situation, we also identified some false-positive comments that are discussed below.

Table 7.8 Comments by Types of TD.

Types of TD	**# of comments ArgoUML**	**# of comments JFreeChart**
Code Debt	490	574
Defect Debt	198	162
Design Debt	160	195
Architecture Debt	114	4
Build Debt	112	4
Requirement Debt	108	34
Documentation Debt	5	0
Test Debt	4	0

Patterns having a bad impact on comments detection that report TD items.

We analyzed the comments and the patterns responsible for their selection as false positives. Table 7.9 presents the main identified patterns for each explored OSS project.

As it is clear from Table 7.9, we see that these six patterns were responsible for 28.70% of false positive comments in Argouml and 52.56% in JFreeChart. This result shows that these patterns are not useful to detect comments that describe a TD situation. Following, we present example comments of each pattern:

Note

The *eXcomment* selected these candidate comments (#1158 and #569) because of the presence of "*NOTE*", but the comment only gives an overview of the information about the code implementation. The tag "*NOTE*" in general represents a note in the code by describing it in more details or revealing relevant facts or ideas. Thus, this pattern is not useful to identify TD items. This trend is in line with what was reported in *FindTD III* - Chapter 6. That is, the pattern "*NOTE*" alone is not decisive to identify a TD item.

Table 7.9 Patterns responsible for many false positive comments.

Patterns	Percentage of false positive comments identified	
	ArgoUML	JFreeChart
NOTE	4.84%	2.11%
necessary	1.85%	2.45%
required	-	13.29%
may be	3.50%	2.54%
can be	8.5%	14.47%
will be	9.67%	17.70%
TOTAL	**28.70%**	**52.56%**

```
Comment #1158 (Argouml)-Class:  ProjectImpl.java
/* NOTE: This was named Project until 0.25.4 when it was replaced by
an * interface of the same name and renamed to ProjectImpl.*/

Comment #569 (jFree){Class:  DateAxis.java * Note:  if the
<code>autoTickUnitSelection</code> flag is * <code>true</code> the
tick unit may be changed while the axis is being * drawn, so in that
case the return value from this method may be * irrelevant if the
method is called before the axis has been drawn.
```

Necessary

The adjective "*necessary*" suggests something required to be done or considered essential in the code. Instead, it simply completes a constructor explanation and a method instructions (Comments #845 and #240).

```
Comment #845 (Argouml){Class:  WizStepConfirm.java
/* The constructor.  Since this constructor does not set the *
necessary instructions, it is private.*/

Comment #240 (jFree){Class:  Axis.java * Constructs an axis, using
default values where necessary.  required to be done
```

Required

The pattern "*required*" was responsible for selecting both comments (#323 and #849) as TD comments. However, they do not represent a TD situation. This pattern was responsible for selection of many false positive comments in JFreeChart (13.29%). This result is a counterpoint to the result presented in *FindTD III*, which shows "*required*" as a decisive pattern to identify a TD item. We will perform new studies to analyze this contradiction.

```
Comment #323 (Argouml) - Class:  AxisSpace.java
/*** A record that contains the space required at each edge of a
plot.*/

Comment #849 (Argouml) - Class:  LogAxis.java
/*** Adjusts the axis range to match the data range that the axis is *
required to display.*/
```

May be and Can be

Both patterns made *eXcomment* selects comments as TD comments, but the content on the comments reports only a "*possibility*" to achieve some objective. However, these patterns alone were not able to detect good comments that really describe a TD situation. This result is in line with individual scores identified and discussed in *FindTD III*. The scores pointed out that this pattern is not so important to identify TD items. We present some examples below:

```
Comment #1688 (Argouml)
Class:AbstractMessageNotationUml.java
/*** Count the number of successors of the given Message.  <p> **
Successors have the same Activator as the given message.  * This
Activator may be null.

Comment #2723 (jFree) - Class:  JFreeChart.java
// if the flag is being set to true, there may be queued up changes...


Comment #420 (Argouml) - Class:  Critic.java
/** * Make this critic active.  From now on it can be applied to a *
design material in critiquing.*/

Comment #1429 (JFreeChart) - Class:  BlockParams.java
/*** A standard parameter object that can be passed to the draw()
method defined * by the @link Block class.*/
```

Will be

The pattern "*will be*" is interpreted as an "*open task verb*" and as a pattern that points out to a task that needs to be done. However, it just makes a promise or a prediction. So, the comments selected by *eXcomment* because of this pattern, in general, do not describe a TD item. The pattern "*will be*" is the most impactful pattern to select false positive comments in both projects. 9.67% of false positives in Argouml and 17.70% of false positives in JFreeChart were returned because of these patterns. The overall observed trend shows that the comments selected because of pattern "will be" describe the methods behavior, classes, variable or feature in the code (comments #946 and #842).

```
Comment #946 (Argouml) - Class:  ConfigurationKeyImpl.java
/*** This class provides definition and manipulation of configuration
keys.  * All keys in the configuration system will be accessed using
the * ConfigurationKey wrapper.

Comment #842 (JFreeChart) - Class:  LogAxis.java
/** * Returns the number format override.  If this is non-@code null,
* then it will be used to format the numbers on the axis.  ...  */
```

Besides these top patterns, we identified some patterns that return good comments to identify TD items, but at the same time, they are responsible for some false positive comments. For example, "*to be implemented*" is a pattern indicating something that still needs to be implemented, pointing to a requirement debt. However, it can also describe interfaces needing to be implemented, representing an instruction on the organization of the program. We present two examples (one false positive and one TD comment) in the following:

```
TD comment:  Comment #9896(Argouml) { Class:  FigMessage.java
/* This next argument may be used to switch off * the generation of
sequence numbers - this is * still to be implemented.  ...  */

False positive:  Comment #11220 (Argouml) { Class:
XmiExtensionWriter.java
/** * An interface to be implemented by classes outside of the model
subsystem that * wish to inject data into the XMI output stream.  */
```

To solve this issue, we need to create a heuristic to analyze whether the comments selected by pattern "*to be implemented*" are describing an interface to be implemented. If the comment describes an interface, it should not be selected by *eXcomment* as a TD comment.

Another point is that software projects have their own vocabulary. That is, software projects have words or expressions that compose names of classes, methods, interfaces, etc. These patterns make our tool classifies several comments as TD item, while they just express information about classes, methods, interfaces, features, etc. Some examples are:

Pattern "Clone"

The pattern "*clone*" can indicate a duplicate code, pointing out to a code debt or design debt. But in the following comments, clone is a feature of JFreeChart so it does not really describe a duplicate code. Thus, the "clone" is not a good pattern to select comments that describe a TD situation in JFreeChart. This case returns many false positive comments in JFreeChart (9.31% of the false positives in JFreeChart). We present some examples below:

```
Comment #370(JFreeChart) - Class:  CategoryAxis3D.java
/** * Returns a clone of the axis.  * * @return A clone.  * * @throws
CloneNotSupportedException If the axis is not cloneable for * some
reason.*/

Comment #7038(JFreeChart) - Class:  StandardChartTheme.java
/** * Returns a clone of the drawing supplier for this theme.  * *
@return A clone of the drawing supplier.*/
```

Pattern "Dependency"

The pattern "*dependency*" describes a state of relying on something for something, especially when this is not normal or necessary. This pattern indicates architecture or build debt, but in the comments presented below, the pattern describes a feature of the software or some information about a class. In Argouml, the word "*dependency*" composes the name of several classes, for example, DependencyResolver.java, FigDependency.java, TestDependencies.java, etc. Thus, the "dependency" is not a good pattern to select comments that describe a TD situation in Argouml because, in general, it returns comments describing some feature of these classes. This case returns 4.63% of false positive comments in Argouml. Above, we presented two examples of these comments.

```
Comment #6240(Argouml) - Class:  FigDependency.java
* This class represents a Fig for a Dependency.  * It has a dashed
line and a V-shaped arrow-head.  * * @author ics 125b course, spring
1998*/

Comment #13791(Argouml) - Class:  DependencyConstraint.java
/** * The <code>DependencyConstraint</code> class is a constraint that
tests * whether two package-dependency graphs are equivalent.  * <p> *
This class is useful for writing package dependency assertions (e.g.
JUnit).
```

Pattern TODO

Lastly, we analyzed the pattern "*TODO*". Developers leave *TODOs* to communicate with other team members that something needs to be done in the future (an open task) and that they were aware of the issue. However, *eXcomment* selected some comments having the pattern "*TODO*" that do not describe open task and TD item.

We found that in the comments #412 and #831, the term "*TODO*" is not the tag "*TODO*", but rather components named "*todo items*" and "*todo panel*" in Argouml. Thus, in Argouml some comments having the pattern "*TODO*" were indicated as false positive comments. The whole list of the false positive comments is available at [30].

```
Comment #412(Argouml) - Class:  Critic.java
// This really creates a lot of to-do items that goes to waste.

Comment #831(Argouml) - Class:  ToDoListEvent.java
/* An appropriate message is shown in case nothing is selected, or *
in case the user selected one of the branches (folders) in the * tree
in the todo panel.*/
```

To make our vocabulary more accurate, we removed the patterns that impacted on the selection of comments by *eXcomment*. After that, we evaluated whether these patterns also impacted on the calculation of final scores. This analysis is presented in the next Section. Finally, we inserted the new patterns identified by the heuristics, resulting in the fifth release of the vocabulary (it is publicly available [30]).

Finding #1: Although the vocabulary allowed to select many comments that describe a TD situation, we identify patterns that are not useful to detect TD items through comments analysis. Besides, we also identify that the software projects have the own vocabulary and it might be responsible for selecting false positive comments.

7.8.2 Do the eXcomment scores represent how much a comment describes a situation of TD situation (RQ2)?

To evaluate this question, we considered scores indicated by the participants and scores calculated by *eXcomment* in two projects, as discussed in Section 3.1, in order to identify how much each comment can describe a TD item.

We adopted two approaches to guide our analysis. First, we analyzed the correlation of the scores using two releases of the contextualized vocabulary, and then we performed comparisons among scores indicated by the participants and the scores calculated by the *eXcomment*. From this analysis, we intend to identify which of the releases better represents the participants' opinion concerning how much the comments describe a TD situation. Next, we calculated the differences between the participant's scores and the *eXcomment*. For this point, we aimed at examining how effectively the participant scores were different or not from scores calculated by the *eXcomment*. More specifically, analyzing comments with high and low differences.

7.8.2.1 Correlation of the final scores. The goal of this topic is to analyze whether the scores calculated by *eXcomment* and participants' scores have a positive correlation. We carried out a Spearman correlation test [97] to examine the relationship. We considered values below 0.3 as weak correlation, values between 0.3 and 0.5 as moderate correlation, and above 0.5 as substantial correlations [98]. Table 7.10 presents the Spearman correlation coefficients in Argouml and JFreeChart projects. The values show a weak and a moderate positive correlation in both projects.

Table 7.10 Spearman correlation coefficients by projects.

	Argouml	**JFreeChart**	**Joined projects**
Spearman	0.1676	0.3647	0.2517
Correlation	Weak	Moderate	Weak

Note that in spite of there is not a strong correlation, the participants' scores have the same behavior as scores calculated by *eXcomment* in many cases. We can see example

comments expressing a trend between the participants' scores and the *eXcomment* in Figure 7.3 separated by dashed lines. We observe that the participants and *eXcomment* had the same judgment, considering how much some comments describe TD situations. Thus far, we have seen that *eXcomment* can calculate final scores indicating which comments are clearer to describe a TD situation.

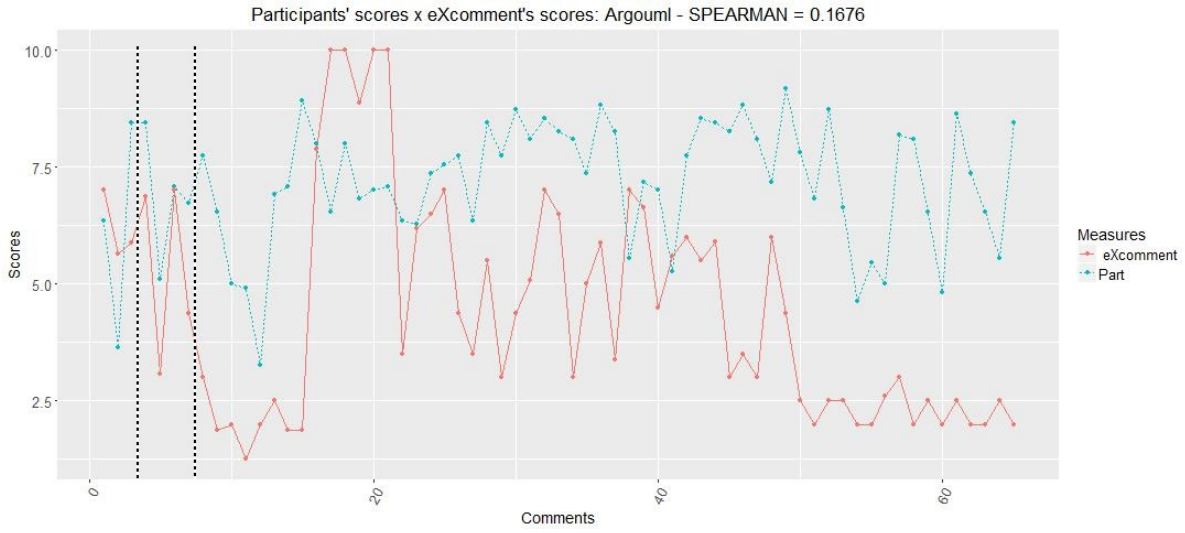


Figure 7.3 Correlation between scores in Argouml.

Finding #2: There is a trend between the participants and *eXcoment*, considering how much some comments describe a TD situation.

On the other hand, we also notice cases where the participant's scores tend to decrease when *eXcomment* scores increase, and vice-versa. We can see some example comments in Figure 7.4 separated by dashed lines.

Besides, these comments affect the correlation and some of them have a significant difference among the scores. In the next Section, we explore some of these comments in order to understand some divergences between the participants' judgment and the final scores calculated by *eXcomment*. Another factor that may be affecting the correlation is the patterns that cause false-positive because these patterns impact on the final scores calculation.

Patterns affecting the correlation.

As we can see from previous Section, there are some patterns that are responsible for many false-positive comments. In order to verify whether these patterns also affected the final scores, we removed them from our contextualized vocabulary and recalculated the final scores. After that, we recalculated the Spearman correlation coefficient.

Table 7.11 presents the new Spearman coefficients. In Argouml, the correlation coefficient went up from 0.1676 to 0.2223 (weak correlation). In JFreeChart, it went up from 0.3647 to 0.4588 (moderate correlation). Also, considering two projects analyzed together, the coefficient went up from 0.2517 to 0.3149 (moderate correlation).

Table 7.11 New Spearman correlation coefficients by projects.

Correlations	Argouml	JFreeChart	Projects together
Spearman - *eXcomment*	0.1676	0.3647	0.2517
Correlation	Weak	Moderate	Weak
Spearman - *eXcomment2*	0.2223	0.4588	0.3149
New correlation	Weak	Moderate	Moderate

Let us consider the following examples:

```
Comment #6555 (Argouml)
Class:  FigNodeModelElement.java
/** Necessary since GEF contains some errors regarding the hit
subject.  *@see org.tigris.gef.presentation.Fig#hit(Rectangle)*/

Comment #7626 (Argouml)
Class:  ActionNewDiagram.java
// TODO: Since there may be multiple top level elements in // a
project, this should be using the default Namespace (currently //
undefined) or something similar
```

The final scores of both comments were changed after the calculation of scores using the new vocabulary. The final score of the comment #6555 (Argouml) changed from 7.0 to 4.0. This reduction occurred due to the removal of the pattern "*necessary*".

In other cases, the final score was reduced due to the removal of the pattern "*may be*". This pattern, in general, was used to describe functionalities of the code, such as description of classes and methods. When the pattern was removed from vocabulary, *eXcomment* also does not identify the heuristics related to it.

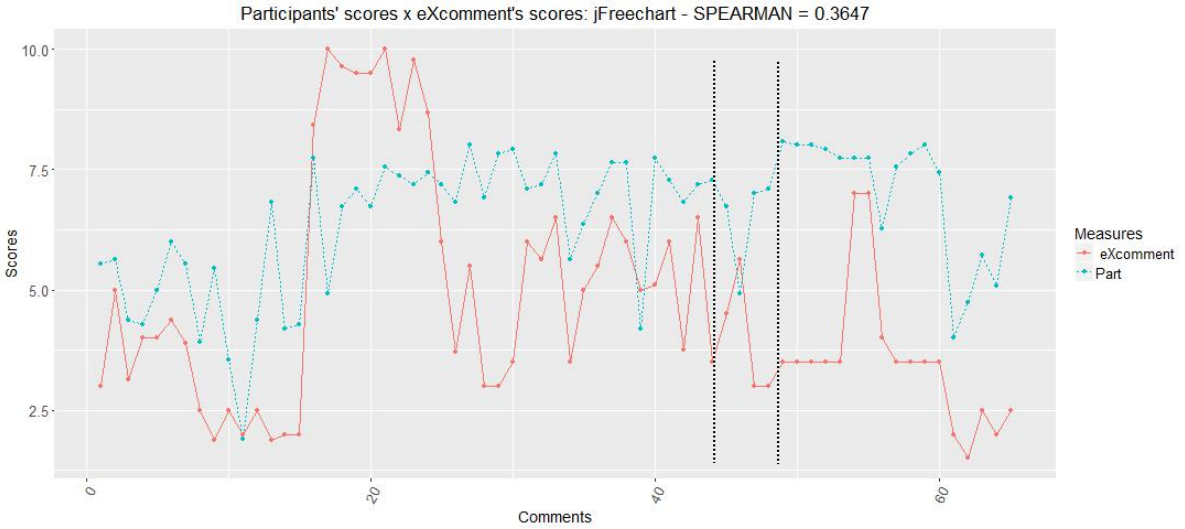


Figure 7.4 Correlation between scores in JFreeChart.

Finding #3: Removing patterns that cause false-positive comments improved the correlation between participants and *eXcomment*.

Following, in order to complement the analysis of correlation, we investigated the differences among the participants' scores and the scores calculated by the *eXcomment* and analyzed the content of some comments.

7.8.2.2 Difference between participants' scores and *eXcomment*'s scores. Figure 7.5 shows a boxplot with differences. The median of differences between participants' scores and the scores calculated by *eXcomment* is 2.70. From this figure, we can see that there are some comments that have significant differences between the participants' values and values of *eXcomment*, but there also are comments with small differences, showing that the participants and *eXcomment* have the same opinion about how much some comments describe a TD situation. We investigated some points of the correlation to understand comments with high and low differences. This allowed us to gather some insights about our approach to identify TD through code comment analysis. First, let us consider some comments with large differences.

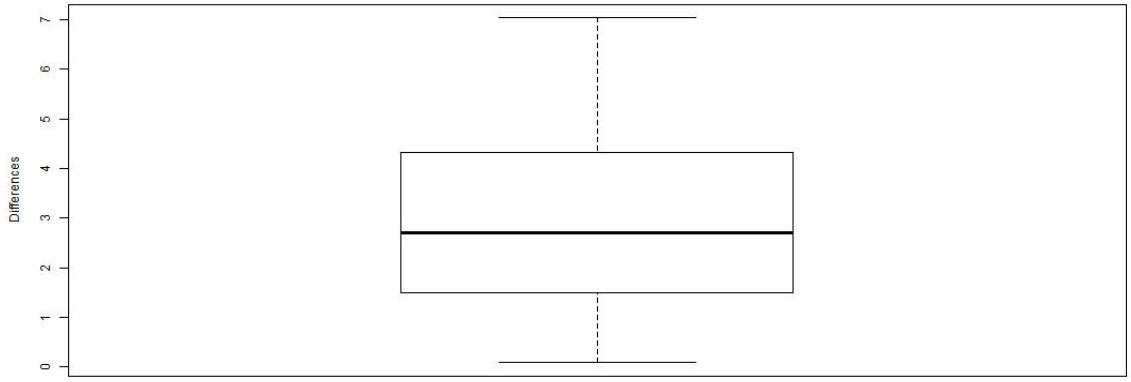


Figure 7.5 Differences between participants' scores and *eXcomment*'s scores.

Heuristics need to be improved

In the example below, the author is certain that the method will fail. Participants assumed that this situation points clearly to a TD item. Even though *eXcomment* has identified the patterns "*TODO*" and "*fail*", the difference among the scores is 4.35. The median of the participants is 8.75 and the score calculated by *eXcomment* is 4.40. *eXcomment* has a heuristic that combines patterns, like a "*Tag + action verb*" (e.g., *TODO: fail*) in order to calculate the final scores, but in this comment, we can observe another heuristic: "*Tag + modal verb + action verb*" (*TODO: will fail*). This case shows that new heuristics can be created to identify other combinations of patterns and improve the final scores calculation.

```
Comment #1633 (Argouml)-Class:  NotationUtilityJava.java
TODO: This will fail with nested Models
```

The comments #8033 and #135 express doubt or uncertainty about features or implementation of the code. *eXcomment* has a heuristic to identify some types of doubt in comments, but it did not interpret these cases as a question or uncertainty. So, the final score was calculated based only on the pattern "*TODO*" (2.00), whereas participants had median value 5.45. Although the heuristics identify several cases, there are situations where developers write a comment in a specific way.

```
Comment #8033 (Argouml) - Class:  TabProps.java
// TODO: No test coverage for this or createPropPanel?  - tfm

Comment #135 (JFreeChart)
Class:  XYDataImageAnnotation.java
// TODO: rotate the image when drawn with horizontal orientation?
```

The next example comment describes a situation where a class implements a method by mistake, but it remains in the project. Participants understood that the comments clearly describe a TD situation. Although the pattern "*mistake*" is identified and it is considered decisive to identify a TD situation, *eXcomment* did not consider the whole chunk of text "*implements PublicCloneable by mistake*". So, the final score was calculated considering only the pattern "*mistake*". *eXcomment* has some heuristics identifying other TD contexts around the identified patterns, but these heuristics did not cover all cases. Participants pointed out this comment with median value 7.80 and the final score calculated by *eXcomment* was 3.00. The difference in this comment is 4.80.

```
Comment #2896(JFreeChart)
Class:  StandardXYSeriesLabelGenerator.java
* This class implements <code>PublicCloneable</code> by mistake...*/
```

The Comment #6983 describes a strange situation and it was pointed out by the participants with median value 4.90. Considering the participants' scores, the comment should be classified in medium level. However, the score calculated by *eXcomment* was 10.00. This happened because of the pattern "temporary", which can describe a temporary solution, for example. But, in this case, it describes only a directory to be used by the method. So, the final score was calculated with high-level value because of the amount of the term "temporary" found in the comment.

This is a case in which semantic is very important to calculate the final scores. It would be quite helpful for *eXcomment* to consider some words around the patterns that are identified alone, for example, "*temporary*". We observe the following cases: "*temporary method*" should be identified because it can report a temporary solution, but "*temporary IP*" should not be identified because it does not seem to report a TD situation. Another fact we highlight is that the size of this comment and the complexity to detect patterns may confuse both participants and *eXcomment* in the analysis of comments.

```
Comment #6983 (JFreeChart)
Class:  ServletUtilities.java
/** Creates the temporary directory if it does not exist.  Throws
a * <code>RuntimeException</code> if the temporary directory is *
<code>null</code>.Uses the system property <code>java.io.tmpdir</code>
* as the temporary directory.  This sounds like a strange thing to
do but * my temporary directory was not created on my default Tomcat
4.0.3 * installation.  Could save some questions on the forum if it is
created * when not present.*/
```

Finding #4: We find that the heuristics are important to identify semantics around the patterns and may be used to improve the TD identification process. However, we also find that *eXcomment* can be evolved in order to cover semantics that are not being addressed in the current heuristics.

Complex comments can be an obstacle

We can note that in spite of the comment #14204 describes situations that can be considered a TD item, some participants pointed out its scores with score values 1 and 2. The difference in this comment was 3.45. The comment describes the method behavior in the first and third paragraph and an issue is presented in the last one. The comment is larger than general comments analyzed and it describes different subjects. These points may confuse humans and *eXcomment* when they are looking for clues that indicate a situation of TD.

```
Comment #14204 (Argouml)
class:  UMLMultiplicityPanel.java
/** Enforce that the preferred height is the minimum height.  * This
works around a bug in Windows LAF of JRE5 where a change * in the
preferred/min size of a combo has changed and has a knock * on effect
here.  * If the layout manager for prop panels finds the preferred *
height is greater than the minimum height then it will allow * this
component to resize in error.  * See issue 4333 - Sun has now fixed
this bug in JRE6 and so this * method can be removed once JRE5 is no
longer supported.*/
```

We present another comment that can be difficult to interpret. In comment #1886, half of participants and *eXcomment* nominated this comment as a comment that does not describe very well a TD situation . But another half of them understood that the comment describes a TD item. It may happen because the chunk of comment "*This will only work when the OrsonPDF library is found on the classpath*" can indicate a code limitation, but this can also indicate just an instruction for users. So, this comment seems to be ambiguous.

```
Comment #1886 (jFree) - class:  ChartPanel.java
/*** Writes the current chart to the specified file in PDF format.
This * will only work when the OrsonPDF library is found on the
classpath.  * Reflection is used to ensure there is no compile-time
dependency on * OrsonPDF (which is non-free software).*/
```

Detection of several patterns and heuristics may help humans and *eXcomment* to interpret complex comments. Thus, a good contextualized vocabulary may be useful in this task, but creating a complete vocabulary is not an easy task.

Finding #5: The complexity of a comment can be an obstacle to participants and eXcomment interpret how much a comment describes a TD situation.

Our contextualized vocabulary is not a finite set of terms

In the comment #9911, the author recognizes that the code is not optimal and that there is a workaround as a temporary solution. We could notice that in spite of the comment describe a workaround, *eXcomment* calculates the final score based on "*workaround*" pattern (this pattern has individual scores 2.5), whereas participants understood that the comment describes clearly a TD situation (participants scores is 8.75). A possible reason for that is that there is another clue for a TD situation, that is "*code here is not optimal*", and the combination of both patterns in a comment can report a major contextualization to identify a TD item. This means that "*code here is not optimal*" can be used as a pattern of our vocabulary to identify other TD situations. Another reason is that scores of some patterns need to be reviewed.

```
Comment #9911 (Argouml) - Class:  FigMessage.java
// so the code here is not optimal but is the best workaround until
/ ArgoUML can provide its own replacement SelectionManager for //
sequence diagram requirements
```

Finding #6: Our contextualized vocabulary is not a final set of terms, it needs to continue being improved. We find that scores of some patterns need to be reviewed.

Nevertheless, in spite of existing comments with substantial differences, showing a low correlation between participants and *eXcomment*, there are several comments with small differences, showing that in some cases participants agreed with the judgment of *eXcomment*, considering the level of the comments in describing a TD situation.

Table 7.12 presents some comments with small differences, patterns, and heuristics found in each comment. We could identify a contextualization/semantic around the pattern identified through heuristics in these cases.

In comment #427, we observed that a method is deprecated through the heuristic "*NOUN+IS/ARE+ADJ*" (*method is deprecated*). The difference in this comment is 0.70. Whereas, in the comment #7593, besides the pattern "*need to*", *eXcomment* identified the heuristic "*OTV+AV*" (*need to make sure*). The difference in this comment is 0.75. Both comments were classified by participants and *eXcomment* as a comment that describes clearly a TD situation.

Table 7.12 Comments with small differences.

Comment	Differences	Identified Patterns	Individual scores
#427	0.70	deprecated, see bug	3.5, 3,5, +heuristic
#7593	0.75	Need to, duplicate	2.5, 2.5, +heuristic
#6210	0.50	could probably be, later	2.0, 2.0
#8522	0.00	for debugging	2.0
#4385	0.80	can be, take care	3.0, 2.0

```
Comment #427 (JFreeChart) - Class:  CategoryAxis.java
// this method is deprecated because we really need the plotArea when
drawing the labels - see bug 1277726

Comment #7593 (JFreeChart) - Class:  ComparableObjectSeries.java
// need to make sure we are adding *after* any duplicates
```

Finding #7: Final scores can be useful to highlight some comments that can really point out to TD items. The detected patterns support people to decide if a comment describes a TD situation or does not.

Participants and eXcomment had similar judgment

The comment #6210 describes a situation where a developer recognized a limitation in a feature of the software. Both participants and *eXcomment* classified the comments as having a medium clarity to describe a TD situation. Participants pointed out this comment with median value 4.30 and the final score calculated by *eXcomment* was 4.00. *eXcomment* found the pattern "*could probably be*", which can describe a possibility – herein, a feature that can be updated – and the pattern "*later*" (adverbs of time) which tells us when this action can happen. Both participants and *eXcomment* are not sure that there is a TD item in this situation.

```
Comment #6210 (JFreeChart)
Class:  RendererUtilities.java
// here we could probably be a little faster by searching for both
indices simultaneously, but I'll look at that later if it seems like
it matters...
```

The comment #8522 describes a code that is useful for debugging. *eXcomment* identified the pattern "*for debugging*", which can identify a code inserted in software just for debugging. However, it does not seem that it indicates a clear TD situation. Indeed, the participants and *eXcomment* agree that the comment does not report much a TD situation or there is doubt whether it really describes a TD item. Participants pointed out this comment with median value 2.0 and the final score calculated by *eXcomment*

was 2.0. The difference in this comment is 0.

```
Comment #8522 (JFreeChart) - Class:  DateRange.java
/** * Returns a string representing the date range (useful
for debugging).
```

The developer tried to get attention to an inadequate code behavior in the comment #4385. Participants and *eXcomment* had scores around 5.0. The difference in this comment is 0.8. It means that the comment describes a TD situation, but it is not totally clear and easy to understand the TD context. One of the patterns identified in this comment was "*take care*". This pattern can signalize something wrong. Another chunk of text identified was "*can be displayed*", which can indicate "*possibility*". Both patterns identified together in this comment increased the final score of comment.

```
Comment #4385 (JFreeChart) - Class:  PiePlot.java
/** Sets the flag that controls whether or not label linking lines
are * visible and sends a @link PlotChangeEvent to all registered
listeners.  * Please take care when hiding the linking lines -
depending on the data * values, the labels can be displayed some
distance away from the * corresponding pie section.*/
```

The whole list of the comments and their differences is publicly available [30].

Finding #8: In spite of existing cases where the final scores calculated by the *eXcomment* do not represent the participants' opinions, we find that, in many comments, the participants and eXcomment had similar judgment regarding how comments are important to detect a TD situation.

7.8.3 How precise is eXcomment to identify TD themes (RQ3)?

We used the precision values, considering the conformity between *eXcomment* and participants in the TD themes classification. We firstly compared the precision values of themes and types of TD classified by participants and *eXcomment*. Then, we investigated the most identified themes by both participants and *eXcomment*.

Comparison of the precision values

Figure 7.6 provides boxplots for the precision of themes and types of TD having found in both projects. This figure shows that the automatic themes classification is more precise than the automatic classification of types performed by *eXcomment*. We conjecture that the themes are simpler to be identified by an automated tool and humans than the types of TD. In this sense, we investigated the most identified themes by participants in conformity with *eXcomment*.

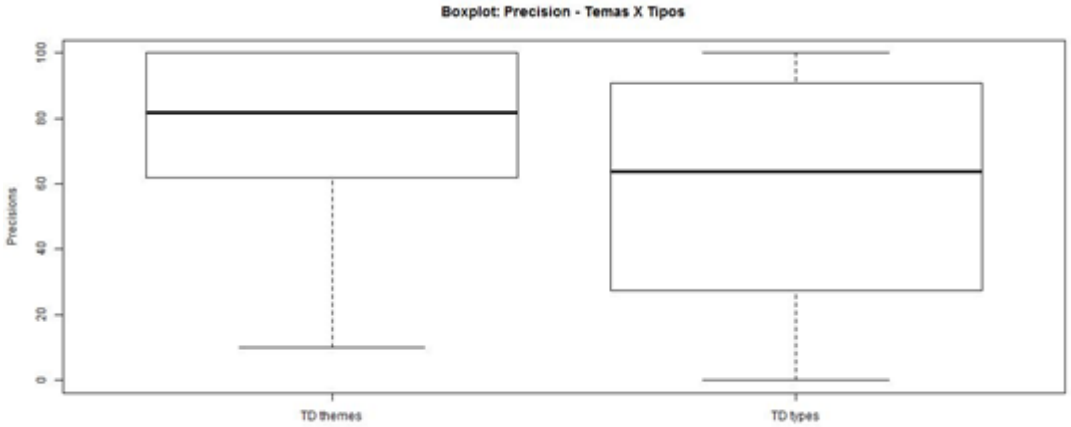


Figure 7.6 Precision values of TD themes and TD types in Argouml and JFreeChart.

Finding #9: Identifying the themes is more precise than identifying types of TD because themes are more related to situation and causes of TD described in comments than the concepts of the TD types.

The most identified themes

Figure 7.7 illustrates the distribution of precision values by TD themes in both analyzed projects. For example, the theme duplicate code was found in four comments from JFreeChart and their precision values vary between 25% and 90%.

The themes "*open task*" and "*deprecated code*" were the most identified themes in Argouml and JFreeChart, respectively. The open task theme, which indicates a task that needs to be done in the future, was classified with precisions higher than 70% in almost all comments in Argouml. This happened because there are many patterns related to tasks to be done (*TODO, need to, must be, etc*) in Argouml. Regarding "*deprecated code*", which indicates codes that will not be further evolved, all comments associated with this theme by *eXcomment* had precision higher than 76% (in both projects), and almost all comments had precision higher than 87% in JFreeChart. That is, participants had a high conformity with our classification in almost all comments considering the theme "*deprecated code*". This happened because there are many patterns related to obsolete or deprecated code (*deprecated, replace the deprecated, deprecated api, etc*) in JFreeChart. Below, we present some examples associated with both "open task" and "*deprecated code*" themes.

Open task

In both comments #23 and #119, the tag "*TODO*" is combined with verbs "*review*" and "*replace*", resulting in the composed patterns "*TODO: Review*" and "*TODO: replace*". The patterns express something to be accomplished in the future, we mean an open task. Both comments had a high precision, considering the "*open task*" theme.

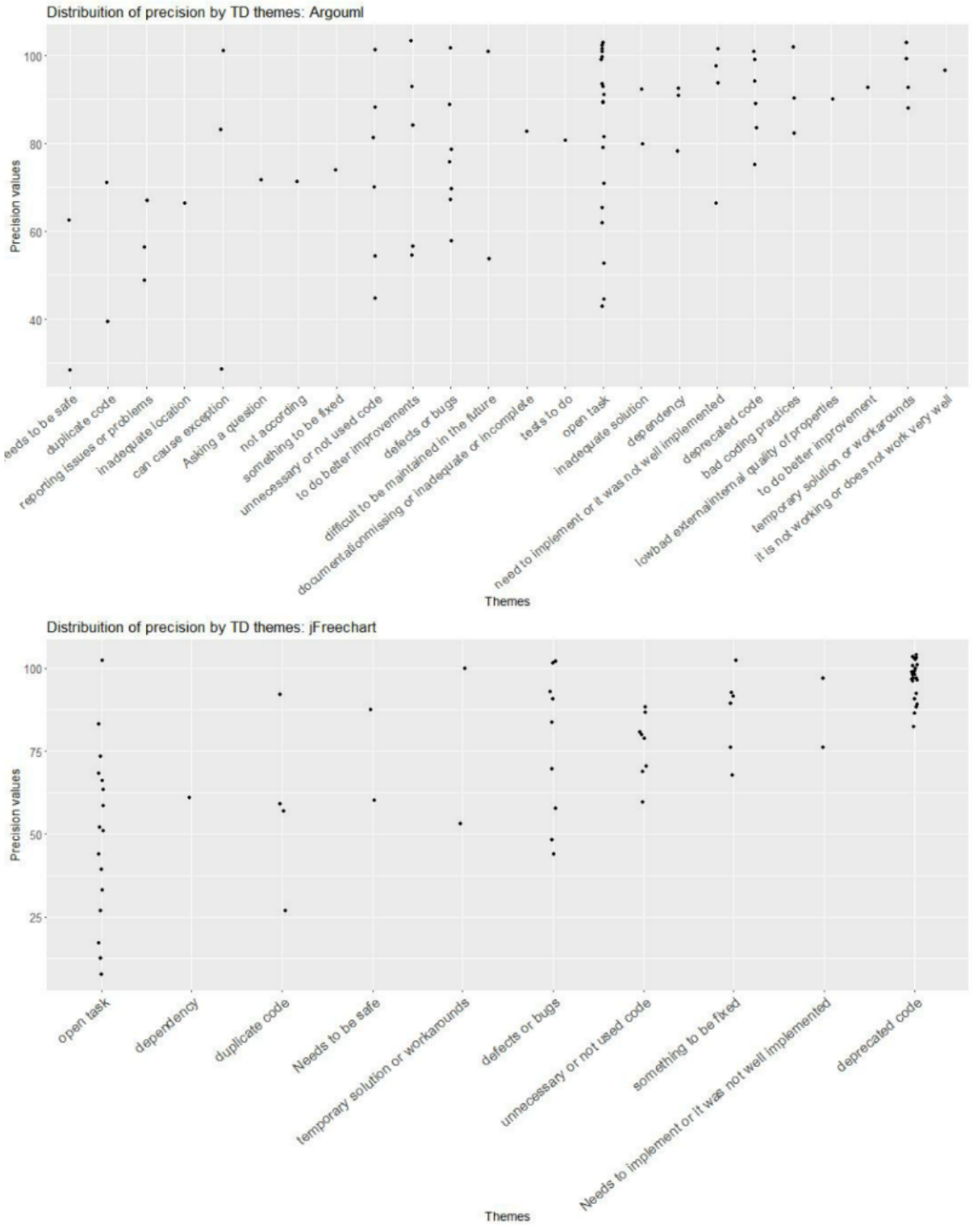


Figure 7.7 Classification of TD Themes identified in *eXcomment* by participants (participants x *eXcomment*).

```
Comment #23 (Argouml) - Class:  AbstractArgoJPanel.java
// TODO: Review all callers to make sure that they localize the title

Comment #119 (Argouml) - Class:  ArgoEventPump.java
* TODO: replace the listener implementation with a EventListenerList *
for better performance
```

Deprecated code

The comments #6280 and #1855 are examples of what we consider as comments related to the theme deprecated code. *eXcomment* identified automatically the patterns "*deprecated*" and "*deprecated call*" in the comments, which express a part of code that is deprecated.

```
Comment #6280 (jFree)
Class:  AbstractXYItemRenderer.java
/*** The item label generator for ALL series.  ** @deprecated
This field is redundant, use itemLabelGeneratorList and *
baseItemLabelGenerator instead.  Deprecated as of version 1.0.6.*/


Comment #1855 (Argouml)
Class:  NotationUtilityUml.java
/* We need to extend the ExtensionMechanismsFactory so that* we can
replace the above deprecated call with something like this:  */
```

Others well-classified themes

Besides, other themes were well-classified by participants in conformity with *eXcomment*, such as: "*defect or bugs*" (Argouml and jFree), "*unnecessary or not used code*" (Argouml and jFree), "*something to be fixed*" (jFree), and "*temporary solution or workaround*" (Argouml). Some examples on these themes are:

Defects or bugs

The comment #2535 was identified automatically as theme "*defect or bugs*" because of the comment chunks "*there is an open bug report*" (bug report) and "*this is wrong*" (*wrong*). The theme had precision value 80%.

```
Comment #2535 (jFree) - Class:  FXGraphics2D.java
/* ...  * According to the Oracle API specification, this method will
accept a * @code null argument, but there is an open bug report (since
2004) * that suggests this is wrong:  ...  */
```

Unnecessary or not used code

In the comment below, the author is certain that a part of the code is "*unnecessary or is not used*", and he suggests removing the fragment from the source code. The comment was associated with this theme because of the pattern "*not used*", and all participants agreed with automatic classification of this theme.

```
Comment #110 (Argouml) - Class:  ArgoEvent.java
/* ...  * TODO: Remove this - not used anyway.  * */
```

Something to be fixed

Almost all participants had the same opinion of our automatic classification considering the theme "*something to be fixed*". The comment #5143 seems to describe that something is not right in the interface and the situation needs to be fixed.

```
Comment #5143 (jFree) - Class:  ArgoEvent.java
// FIXME: the renderer interface doesn't have the drawDomainLine()
method, so we have to rely on the renderer being a subclass of
AbstractXYItemRenderer (which is lame)
```

Temporary solution or workarounds

eXcomment considered that the comment #7265 describes a workaround, indicating that developers implemented a temporary solution to solve some issue in the code. All participants agreed that the comment defines a temporary solution or workaround.

```
Comment #7265 (Argouml) - Class:  TempFileUtils.java
// skip backup files.  This is actually a workaround for the // cpp
generator, which always creates backup files (it's a // bug).
```

Some other themes were classified with good precision, but they were evaluated through few comments, such as "*it is not working or does not work very well*" and "*needs to implement or it wasn't well implemented*", as it can be seen in Figure 7.7. Thus, we need to perform future experimentation focused on these themes to evaluate whether they are really in conformity with our automatic classification. In the following, some of them are presented.

It is not working or does not work very well

All participants agreed with our automatic classification taking into account that the comment #7906 describes something that "*is not working or does not work very well*". The pattern "*not work*" was the responsible for the identification of this theme.

```
Comment #7906 (Argouml)-Class:  SaveGraphicsManager.java
/* The next line does not work:  */
```

Need to implement or it wasn't well implemented

Regarding the comment #9733, all participants also agreed with *eXcomment* considering the classification of the "*need to implement or it was not well implemented*" theme. The automatic classification was possible because the pattern "*not implemented*" has been found in this comment.

```
Comment #9733 (jFree) - Class:  SWTGraphics2D.java
/** Not implemented yet - see
@link Graphics2D#getDeviceConfiguration() ...  */
```

Finding #10: Our strategy worked well to identify automatically themes in the most of the comments analyzed in this experiment through comment analysis. This means we can consider that *eXcomment* is calibrated to identify the most of themes by using a contextualized vocabulary. Thus, it can be used to classify the type(s) of the TD items.

7.8.4 How can TD themes support the TD items classification (RQ4)?

To answer this question, we analyzed the relationship between the precisions of themes and types of TD and analyzed the cases with not good relationship (unbalanced relationship) to support the identification of the types.

Relationship between identified themes and TD types

We investigated how themes are related to the identification of TD types. Figure 7.8 presents the relation between themes and types considering their precisions. From this figure, we can analyze how themes are related to types, considering the automatic classification and the participants' judgment. Themes are illustrated in x-axis and types in y-axis; the precision values of each relation (theme-type) are presented in the intersection of the themes and types, for example, for the relation "*bad coding practices and code debt*" (highlight in figure top - Argouml), we see the values 91-70 (theme-type). This means that the participants agreed with *eXcomment* in 91% of the cases, considering the identified theme. Whereas that the participants classified the comments as code debt in conformity with *eXcomment* in 70% of the cases that were classified in this theme. If a ratio of a theme-type is 100-100, this means that all comments that the *eXcomment* classified as a theme and a type were also classified by the participants.

We consider that the ratio theme-type is balanced and useful to classify TD types whether the precision of a theme is equal or higher 60% and the precision of a type is also equal or higher 60%, for instance, "*deprecated code x code debt: 97-97 in JFreeChart*, and "*difficult to be maintained in the future x code debt: 77-95*" in Argouml. By contrast, if the precision of a theme is equal or higher 60% and the precision of a type is lower 60% (or vice-versa), we consider that the theme-type ratio is not balanced and needs to be further explored in order to be calibrated, for instance, "*duplicate code x design debt: 60-30*" in jFreechar and "*need to implement or it was not well implemented x requirement debt: 89-20*" in Argouml.

Next, we discuss some cases focusing on the ratio theme-type we consider not balanced in order to try to understand the divergent classifications.

Not balanced relation

The relation "*dependency x architecture debt*" is considered not balanced in both projects. This result shows that the "*dependency*" theme had the precision value 88% in Argouml and 60% in JFreeChart, but the architecture debt had only precision value 15% in Argouml and 0% in JFreeChart. To put it another way, *eXcomment* identified

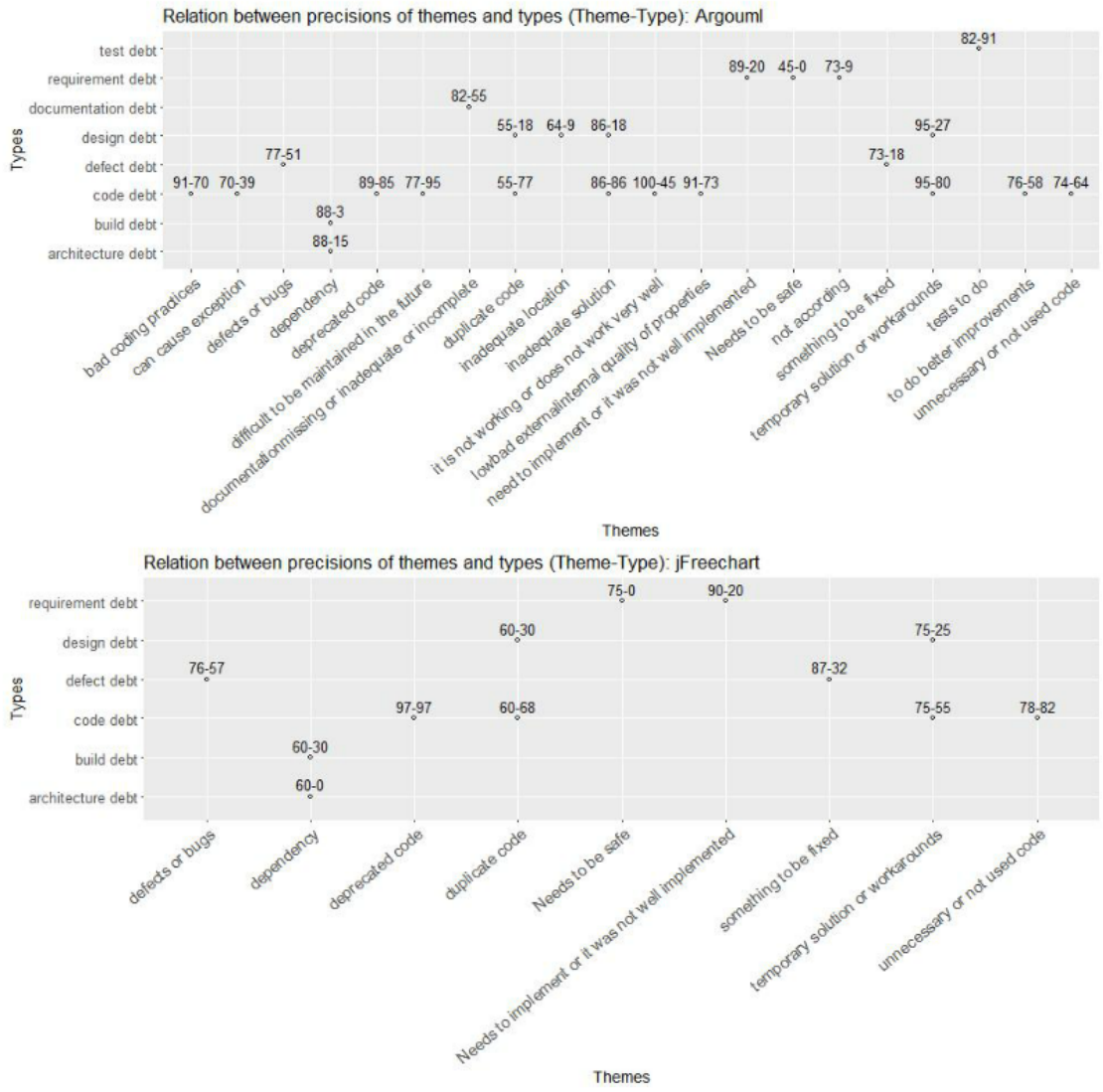


Figure 7.8 Classification of themes and types in Argouml and JFreeChart.

automatically comments as architecture debt because of this theme, participants agreed with the classification of the theme but they disagree with the classification of architecture debt; we mean, this theme seems not to support the identification of the type architecture debt. Another ratio considered not balanced in both projects is "*dependency x build debt*". *eXcomment* also uses the "*dependency*" theme to classify comments as build debt. This ratio is considered not balanced because the precision values are 88-3 in Argouml and 60-30 in JFreeChart.

An example comment classified as *dependency x architecture debt* and *dependency x build debt* is in the following.

```
Comment #989 (Argouml) - ArgoModeCreateFigLine.java
// TestDependencies thinks this creates a dependency cycle
```

This is an example of what *eXcomment* associated with the theme "*dependency*", and classified as architecture and build debt. Although the comment refers to a cyclic dependency (which can affect architectural requirements and make), only one participant classified it as "*dependency x architecture debt*" and nobody classified it as build debt. It is worth considering that the participants classified this comment as code, design, defect, and test debt. For them, "*dependency*" theme may be more related to another type of TD than architectural and build debt, and, thus, the participants got confused about what is an architectural and build debt. This result is in line with what we discussed in the previous section.

In the previous topic, we noticed that requirement debt did not have a good precision (all comments classified as requirement debt had precision lower than 50%). This result is confirmed by analyzing the themes related to requirement debt. All relationships of requirement debt were considered not balanced in both projects. Thus, we analyzed some of these cases below.

The "*need to be safe*" was considered the worst theme to identify requirement debt with 45%-0 in Argouml and 75%-0 in jFree. In both projects, nobody, who classified the comments as "*need to be safe*", classified them as requirement debt. This shows that comments classified as this theme do not describe a situation of requirement debt.

Finding #11: Some relations themes-types are calibrated and useful to support *eXcomment* and participants to classify correctly TD types. However, we also find that other themes seem not to support the identification of some TD types. As a consequence of this, some relationships between TD themes and TD types need to be reviewed in order to be useful for identification of TD types.

Other two themes that were not considered important to identify requirement debt are "*need to implement or it was not well implemented*" (89%-20% for Argouml and 90%-20% for jFree) and "*not according*" (73%-9% for Argouml). Participants understood that comments associated with both themes by *eXcomment* describe something that needs to be implemented or it was not well implemented and something is not according to specification. However, although the comments refer to the lack of synchronism between the requirements specification and what is currently implemented, participants did not classify the comments as requirement debt. A possible reason is that comments describing something about implementations can easily be confused with code debt or the participants did not understand the concept of requirement debt type.

We can also realize that all relationships of themes-design debt were considered not balanced in both projects. The result indicates that all themes (duplicate code, inadequate location, temporary solution or workarounds, and inadequate solution) related to design debt were not considered by participants as good indicators to support the identification of the types. Conversely, all these themes are also related to code debt, having good and balanced relationships. For example, "*inadequate solution x design debt*" relationship had precision 86%-18%. Whereas "*inadequate solution x code debt*" relation

had precision 86%-86%, showing that in 86% of cases that were associated with "*inadequate solution*" theme were also classified as code debt by the participants. An example comment is presented below:

```
Comment #2298 (Argouml) - Class:  PrivateHandler.java
// There are probably better ways to implement this than using na //
ArrayList of Hashtables.
```

The comment #2298 describes an implementation that can be possibly an inadequate solution and suggests a solution that can be improved using other strategies. *eXcomment* classified this comment as design and code debt because of the theme "*inadequate solution*", but only a participant classified it as design debt and the other nine participants classified it as code debt.

Another case is the relationship "*temporary solution or workarounds x design debt*". The comments associated with this theme are also classified as design and code debt, but this theme was not considered by participants as good indicators to support the identification of design debt. On the other hand, "*temporary solution or workarounds*" theme was considered useful to support the identification of code debt with ratio 95%-80% in Argouml.

Next, we analyze the classification of each type of TD by *eXcomment* and participants considering the precision values and a qualitative investigation.

Finding #12: A theme can be associated with different types of TD. However, in some cases, a theme describes a type of TD more clearly than others.

7.8.5 How precise is eXcomment to identify types of TD (RQ5)?

We used two approaches to address this research question. The first considered types identified automatically by our approach. The main goal of this analysis is to identify the most precise types of TD classified by the participants in concordance with *eXcomment*. In the second, we ran an analysis of the comments that our approach did not identify any type of TD (no type). Herein, the main goal is to identify and analyze the cases where the participants classified some types of TD but *eXcomment* was not able to identify any of them. We investigated why *eXcomment* did not classify these comments in any type of TD. In both approaches, we considered the precision values for each comment analyzed in this experiment concerning the participants.

7.8.5.1 Types identified automatically by our approach. Figure 7.9 shows a box plot illustrating the precision distribution by each type of TD identified by *eXcomment*. Precision indicates how many types of TD classified by participants were also classified by *eXcomment* (see Equation 7.1 in Section Analysis Procedure). The figure in the top illustrates types analyzed in comments from the Argouml and the figure in the bottom

illustrates the types in JFreeChart. From this figure, we can identify the types that were more classified simultaneously by participants and our approach.

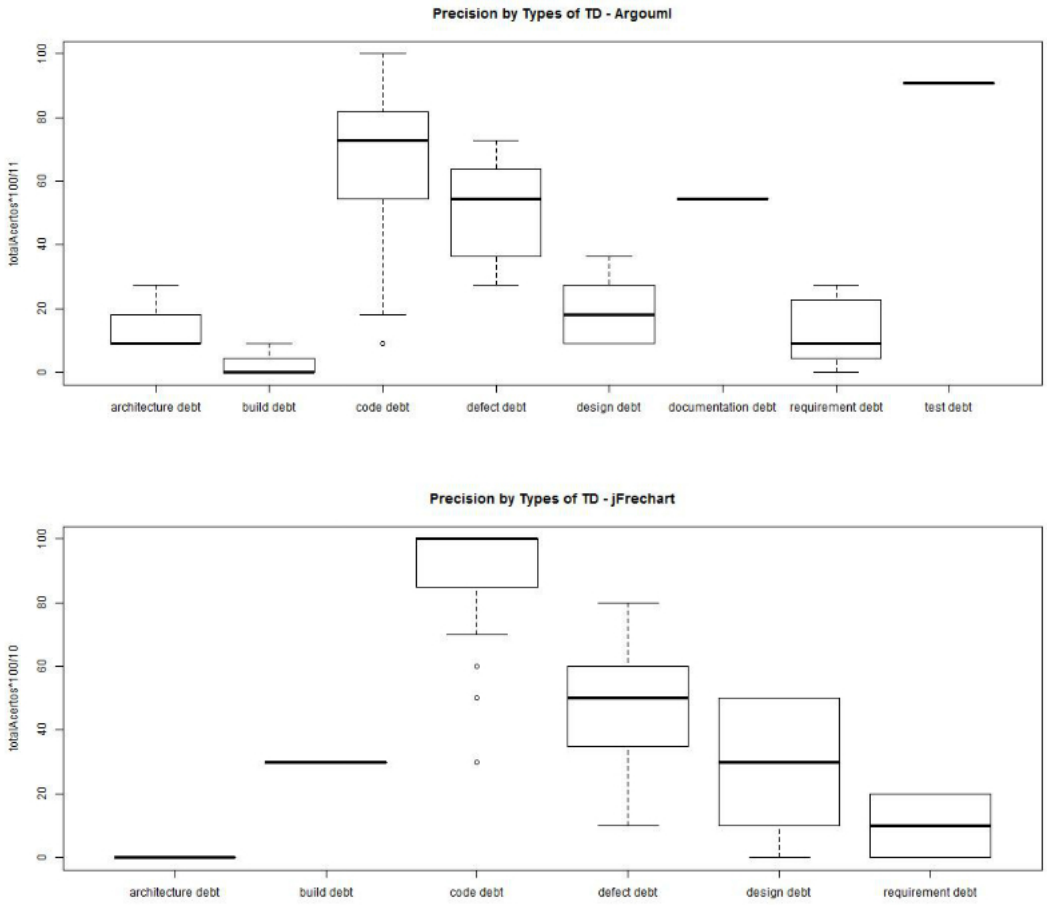


Figure 7.9 Analysis of TD types by participants, considering the automatic detection of the types by *eXcomment*.

We can see that the most identified type was code debt, and the second one was defect debt in both projects. The median values of precision are 72.73% for code debt and 54.54% for defect debt in Argouml, whereas for jFree, the median is 100% for code debt and 50.00% for defect debt.

Additionally, we also performed a hypothesis test to reinforce the analysis of code debt as the main TD type described in code comments. To do so, we defined the following null hypothesis:

H0: The precision between code debt and other types of TD identified by participants in conformity with our approach (eXcomment) are equal.

We ran a normality test considering two groups (a group with code debts, and another group that clustered the other types of TD), Shapiro-Wilk and identified that the distribution is not normal, as it can be seen in Table 7.13. After that, we ran the Mann-

Whitney test, a non-parametric test, to evaluate our hypothesis. We used a typical confidence level of 95% ($\alpha = 0.05$). As shown in the same table, the p-value calculated (p = 2.2e-16) is lower than the α value. Consequently, we may reject the null hypothesis (*H0*).

Table 7.13 Hypothesis test for analysis of code debt type.

	Shapiro-Wilk (Normality,Test)		**Non-parametric Test**
	Code debt	**Other types**	**Mann-Whitney**
p-value	7.622e-07	0.0002861	2.2e-16

We also evaluated our results in terms of magnitude, testing the effect size measure. We calculated Cohen's d [91] in order to interpret the size of the difference among the distribution of the types. We used the classification presented by Cohen [91]: 0 to .1: No Effect; .2 to .4: Small Effect; .5 to .7: Intermediate Effect; .8 and higher: Large Effect.

The magnitude of the result ($d = 2.229$) also confirmed that there is a difference (Large Effect) on the precision values concerning the types. This evidence reinforces the alternative hypothesis and shows that the results were statistically significant.

We also noted that test and documentation debt had good precisions but only one comment classified in each type was explored in this experiment. Thus, even considering that the precisions on the comments above are relevant, we do not have enough data to perform a suitable statistical analysis.

Finding #13: Code debt is the most identified TD type by the participants in conformity with the *eXcomment*.

Code debt comments are those that describe problems related to source code which may make it more difficult to be maintained and evolved. Following, we present some example comments that were classified as code debt by both participants and *eXcomment*.

```
Comment #427 (jFree) - Class:  CategoryAxis.java
// this method is deprecated because we really need the plotArea when
drawing the labels - see bug 1277726.
```

eXcomment classified the comment #427 as code debt because of the chuck "*method is deprecated*". Defect debt also was detected because of the pattern "*see bug*". The comment expresses a justification for a deprecated method and reports a bug. Regarding code debt, all participants agreed with our classification.

```
Comment #8092(jFree) - Class:  SubSeriesDataset.java
/* This class will be removed from * JFreeChart 1.2.0 onwards.
Anyone needing this facility will need to * maintain it outside of
JFreeChart.*/
```

The comment #8092 is a clear example of what we consider as an open task related to code that will be done in the future. This comment was classified as code debt by

eXcomment and by 90% of the participants. Only one participant classified it as a build debt.

```
Comment #8842 (Argouml) - Class:  TestProjectSettings.java
/* We no longer send individual events,* so next code is obsolete:
*/
```

The above comment (#8842) reports an obsolete code and it was identified as code debt by *eXcomment* and almost all participants. It was identified as code debt because of the chunk "*code is obsolete*". Even though the author of the comment is assured that the code is obsolete, the code was not updated or deleted, indicating that the code needs to be reviewed.

In spite of the code and defect debt had good values of precision, it is possible to notice that there are some outliers and comments with precisions lower than medians classified in other types. From Figure 7.9, we also notice that the types build, architecture, and requirement have all their comments lower than 50% of precision in both projects. Regarding design debt, only two comments had 50% of precision in JFreeChart and the others are lower than 50%. It is possible to state that the classifications of these types by *eXcomment* are not in line with classifications performed by the participants. Next, we analyzed a sample of comments with precisions lower than medians in order to try to comprehend the divergence between the participants' judgment and *eXcomment*. We found different reasons that may have impacted on the classification of the TD items. In the following, we discuss some of them.

Relation: Patterns x TD types

The comment #7805 had just 9.09% of precision. Only one participant agreed with our approach considering code debt type. This comment was classified as code debt because it was related to the "*rebuild*" pattern. Maybe this pattern is not good to detect code debt.

```
Comment #7805 (Argouml)
Class:  UMLTagDefinitionComboBoxModel.java
// Just mark for rebuild next time since we use lazy loading
```

The comment below (#5786) describes a method that needs to be defined. This comment has a tag "*FIXME*", and maybe the author used the tag to attract attention for the issue. The tag is related to defect debt by the *eXcomment* and because of this relation the comment was classified as defect debt. Its precision is 10%, which means that only one participant classified the comment in accordance with *eXcomment*. All others classified the comment as a code debt type (9 out of 10 participants). This may have happened because the comment described a problem about a method in the code.

```
Comment #5786 (jFree) - Class:  CategoryItemRenderer.java
// CREATE ENTITIES FIXME: these methods should be defined
```

The next comment also has a tag "*FIXME*" like the previous comment. The *eXcomment* also classified this comment as a defect debt ("*something to be fixed*"), but only

two participants pointed out it as defect debt. However, 90% of participants classified the comment as code debt, and 40% of them classified it as a requirement debt.

```
Comment #1024 (jFree) { Class:  PeriodAxis.java
// FIXME: implement this...
```

Maybe the tag "*FIXME*" was used wrongly because both comments do not seem to describe a defect debt. In fact, we consider that a comment having this tag describes a defect debt situation.

Finding #14: Some relationships between patterns and types of TD need to be reviewed.

Comments classified in more than one type of TD

In the comment #619, the author certainly implemented a code that is not the best solution, but it was not possible to do better due to some limitation. The comment was classified as code debt and design debt by *eXcomment*. This comment was associated with three themes by *eXcomment*, such as "*inadequate solution*", "*difficult to be maintained in the future*", and "*to do better improvements*". All these themes are related to code debt and only the theme "*inadequate solution*" is also related to design debt. The precision for code debt is 90.90%. Regarding design debt, only 27.27% agreed with this type.

```
Comment #619 (Argouml) - Class:  ResolvedCritic.java
* This is a rather bad hash solution but with the @link
#equals(Object)} * defined as below, it is not possible to do
better.*/
```

The comment #3089 was classified by *eXcomment* as build and architecture debt because of reports "*dependency*" between two classes and as defect debt because the theme "*something to be fixed*". For our approach, this comment refers to issues that make a task harder, that is it indicates a structural dependency. However, only one participant categorizes the comments as a build debt and three participants categorize it as architecture debt. Regarding defect debt, only two participants agreed with this classification. Thereby, this comment was classified into three different types, showing that when a comment deals with more than one type associated with different themes, it may make the comment more complex for people to choose in which type(s) the comment is classified.

This is another example of comment that reports different issues considering different types of TD. This situation may make it a hard comment to be analyzed by humans and ambiguous when it is automatically classified by a tool.

```
Comment #3089(Argouml)
Class:  LastRecentlyUsedMenuList.java
// TODO: This class is part of a dependency cycle with ProjectBrowser
and // GenericArgoMenuBar, but should be fixed if project open/close
is moved
```

Following, the example comment #9615 was classified as design and code debt by *eXcomment* and participants. The precision values are 50% and 90%, considering design and code debt, respectively. The comment was classified as both design and code types because of the comment chunk "*I decided it was better to duplicate some code*". This fragment of comment reports a decision in that some code was duplicated. *eXcomment* considers duplicate code an indicator of code and design debt because it describes issues related to bad coding practices - complex code, low-quality code, or duplicate code – (code debt) and practices which violate the principles of good object-oriented design (design debt).

Even though the precision of code debt is higher than design debt, we note that some participants understood that the comment describes both types of TD (precision value of 40%). This comment is an example that brings an intersection of concepts of different types of TD (design and code debt).

```
Comment #9615 (jFree) - Class:  YWithXInterval.java
/*** A y-value plus the bounds for the related x-interval.  This
curious * combination exists as an implementation detail, to fit into
the structure * of the ComparableObjectSeries class.  It would have
been possible to * simply reuse the @link YInterval} class by assuming
that the y-interval * in fact represents the x-interval, however I
decided it was better to * duplicate some code in order to document
the real intent.  ** @since 1.0.3*/
```

Finding #15: When a comment describes a situation that can be classified in more than one type of TD, the comment is more complex for participants to decide which type(s) the comment is classified. We also find that some comments describe a TD situation that reports an intersection of concepts of different types of TD. Thus, identifying the type of TD is not easy because some concepts are close to each other.

Deficiency on the heuristics to classify a TD item

The comment #6430 was classified as design debt by *eXcomment* and participants (with a precision value of 50%). Additionally, it was also classified as code debt by *eXcomment* and participants (with a precision of 90%) and defect debt by only one participant, but it was not classified as defect debt by *eXcomment*. Analyzing the comment, we observe that the comment chunk "*This should get fixed at some point*" may indicate a possible defect debt. *eXcomment* identified this fragment of comment through the heuristic "*modal + AV*" composed by the modal "*should get*" and the action verb "*fixed*" (*should get fixed*), but the combination of patterns identified through heuristics are not related to types of TD yet. Thus, *eXcomment* could not identify the defect debt automatically.

```
Comment #6430 (jFree) - Class:StackedXYAreaRenderer.java
/* SPECIAL NOTE: This renderer does not currently handle negative
data values * correctly.  This should get fixed at some point, but the
current workaround * is to use the @link StackedXYAreaRenderer2} class
instead.*/
```

In this comment (#8033), almost all participants classified it as test debt. The author is questioned about the existence of test coverage. As a matter of fact, our approach identified the heuristic "*Tag + AV*" composed by "*TODO*" and "*not test*" (*TODO: No test*), which may indicate a clue of test debt. However, as it was discussed in previous comment, the patterns identified through heuristics are not related to types of TD.

```
Comment #8033 (Argouml) - Class:  TabProps.java
// TODO: No test coverage for this or createPropPanel?  - tfm
```

Even more, *eXcomment* is not able to identify automatically the types of TD in some comments because they only describe a TD item without bringing tips about their types. Another fact is that some patterns and the heuristics are not related to a type of TD. In the next section, we analyzed some comments without types identified automatically by our approach.

Finding #16: Our heuristics are important to identify combination of patterns but they are not prepared to classify the type of TD.

7.8.5.2 Comments without types identified automatically by our approach.

Figure 7.10 illustrates distributions of precision values for comments that *eXcomment* did not identify any type of TD. From this figure, we notice that the majority of the precisions are below 40% in both projects, and there are comments with zero precision value. This means that although *eXcomment* did not detect any type of TD for these comments, participants (in general) classified them into some type(s).

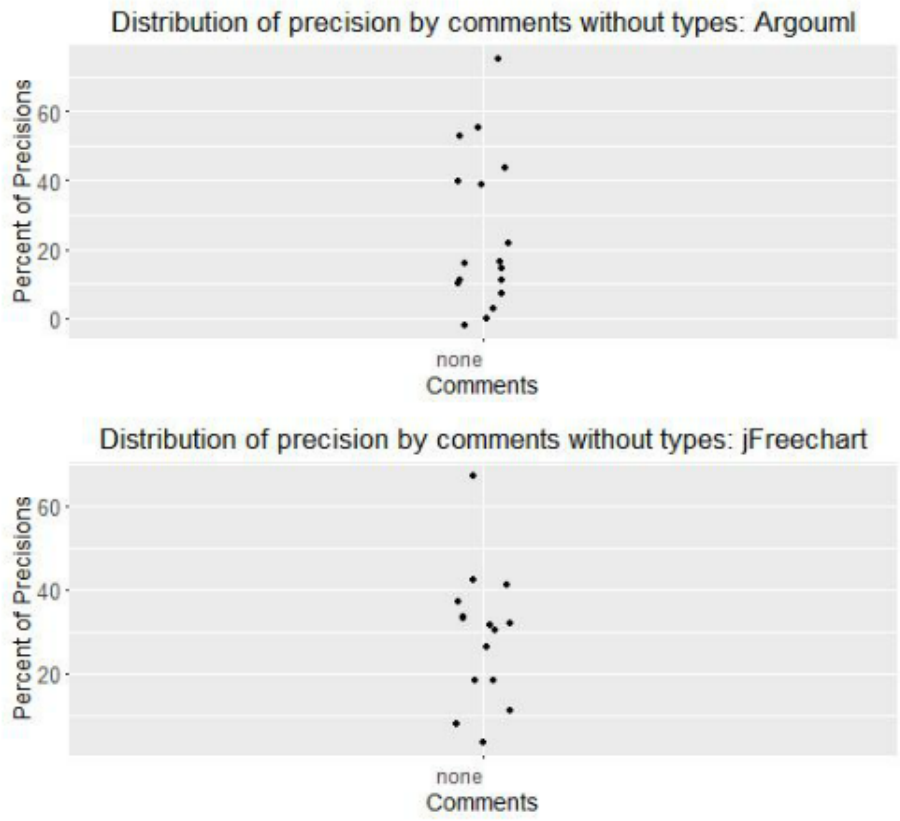


Figure 7.10 Analysis of precisions by comments without types of TD (Argouml and JFreeChart).

Figure 7.11 presents a bubble chart where each bubble represents the number of times that a type was chosen by participants in each comment. From this figure, it is possible to highlight that participants labeled these comments in different types. However, code debt is the most chosen type by the participants in these cases. Last but not least, we noted that some comments are out of this trend, such as the comments #8033 and #8936. They were mainly classified as test debt type by the participants. The comments #10994 and #8009 were mainly classified as documentation debt.

Next, we examined a sample of these comments in order to explore the disagreement between *eXcomment* and participants, considering the classification of types of TD.

The comment in the following brings information about different issues, such as "*this approach should not be used*" (Unnecessary or not used code), "*the current implementation is incorrect*" (It is not working or does not work very well), "*TODO: What event names?*" (Asking Question theme), and "*not really well documented*" (Documentation-missing or inadequate or incomplete theme). *eXcomment* identified both two first cases through the identifications of the heuristics "*OTV+AV*" and "*Noun+is/are+adj*", but it could not automatically classify them in a TD type due to the reason discussed previously.

In comment #3601, only three participants rated it as no type (could not classify the comment in any type of TD). That is, other participants classified the comment in some

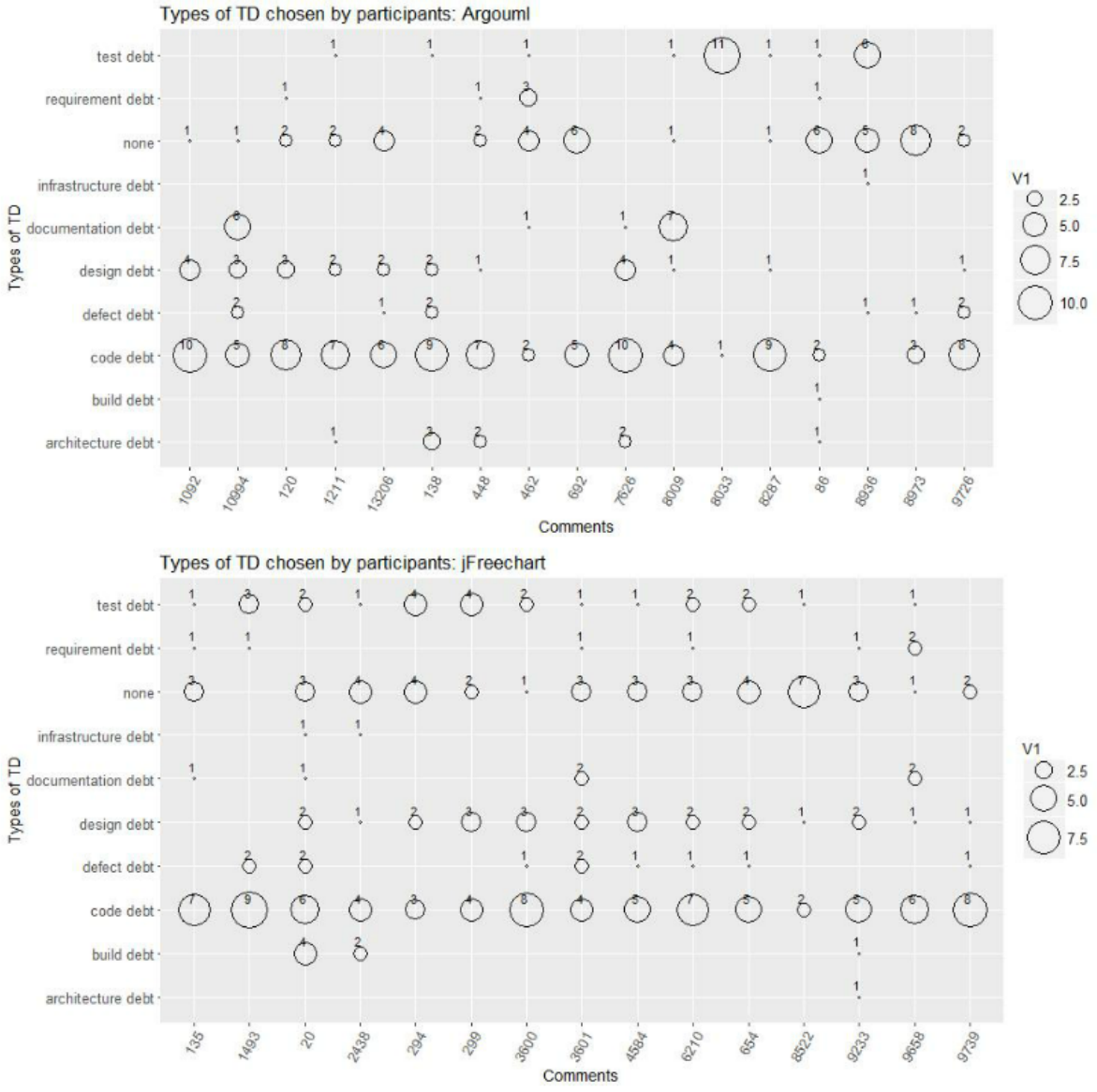


Figure 7.11 Classification of types of TD by participants in comments that our approach did not identify any type of TD.

type of TD: code (4), defect (2), design (2), documentation (2), requirement (1), and test (1). We did not consider this comment easy to choose a type of TD because it does not have a clear clue related to a specific debt.

Comment #10994 (Argouml) - ModelEventPump.java

```
/** ...  * Since the garbage collecting mechanism is not really
deterministic * a forgotten about listener might still receive events.
Unless it can * handle them in a harmless way, this approach should
not be used.  * TODO: (Is this still true or does it refer to the
NSUML * implementation?  - tfm 20051109) * (This is part of the
contract that is established between the Model * subsystem and its
users.  If that is not fulfilled by the current * implementation, then
the current implementation is incorrect.* Linus 20060411).<p> * TODO:
What event names?  * The event names generated are @link String}s
and their values and * meanings are not really well documented.  In
general they are the * name of an an association end or attribute in
the UML metamodel.<p> * * Here are some highlights:<ul> * <li>"remove"
- event sent when the element is removed.  * </ul> * * @author Linus
Tolke*/
```

Comment #3601 (JFreeChart) - ContourPlot.java

```
(jfree) - // do we need to update the crosshair values?
```

Next, we can find another example of comment classified in almost all analyzed types (seven out of nine types) by the participants, but did not associate with any type by *eXcomment*.

This comment seems to be hard for participants to decide which type(s) of TD is(are) related to it. They classified it in the following types: build (4), code (6), defect (2), design (2), documentation (1), infrastructure (1), none (3), and test debt (2). The requirement and architecture debts were the unique types that were not chosen by the participants.

Regarding *eXcomment*, it identified only the pattern "*WARNING*" here. However, there are other comment chunks that may bring us clues regarding which types of TD the comment is describing, such as "*THIS CLASS IS NOT PART*", "*NOT RELY ON THIS CLASS*", and "*SUBJECT TO ALTERATION OR REMOVAL*". These expressions may be related to a type of TD and inserted in our vocabulary in order to recognize cases similar to this comment.

Comment #20(jFree)
Class: XYSmoothLineAndShapeRenderer.java

```
//* ...  * WARNING: THIS CLASS IS NOT PART OF THE STANDARD JFreeChart
API AND IS * SUBJECT TO ALTERATION OR REMOVAL. DO NOT RELY ON THIS
CLASS FOR * PRODUCTION USE. Please experiment with this code and
provide feedback.*/
```

To conclude, we present a case (#9739) classified as no type by *eXcomment* and as code debt by almost all participants.

The comment describes a hint in the code for a better solution in handling some methods. However, *eXcomment* did not identify a type of TD for this comment because there are no obvious patterns that characterize its type. We can see that the author

referred to methods using their names (setPaint (), setColor (), setColor () and getPaint ()), and this semantic can be an obstacle for a tool to classify automatically the comments as code debt type.

```
Comment #9739 (jFree) { Class:  SWTGraphics2D.java
// TODO: it might be a good idea to keep a reference to the color
specified in setPaint() or setColor(), rather than creating a new
object every time getPaint() is called.
```

Finding #17: Some comments are not easy to choose a type of TD because they do not have clear clues related to a specific type of debt. *eXcomment* did not classify some comments because it did not identify some expressions describing a type of TD. Thus, new patterns need to be identified and related to a TD type.

7.9 SUMMARY OF FINDINGS AND INSIGHTS

Thus far, we have seen that our approach can perform well to automatically identify and classify TD items. However, there are some insights we present below that deserve further investigations.

7.9.1 Analysis of patterns which impacted on selection of false positive comments.

We found some patterns that were responsible for a large number of false positive comments and they are not useful to identify TD items through code comment analysis. Removing these patterns from our vocabulary may have improved the accuracy of our approach and reduced the false positive comments in 28.70% for Argouml and 52.56% for JFreeChart. This could have improved the selection of comments related to the TD description.

Another aspect we analyzed was some patterns returning good comments to identify TD items, but at the same time, they are responsible for some false positive comments. We identified that software projects have their own vocabulary and it might be responsible for selecting false positive comments. To mitigate this issue, we need to know previously the project vocabulary to isolate the patterns in order to reduce the false positives selected by *eXcomment*.

We want to widen the analyses in order to investigate the existence of other patterns that can impact on detection of TD items. Thus, there are some cases that deserve further examination. A significant contribution is that these analyses are important so that in future works we develop more advanced heuristics aiming to reduce false positive comments.

7.9.2 Analysis of final scores.

We compared the participants' scores and the scores calculated by the *eXcomment*, analyzing the correlation coefficient between scores and the differences between them. The

main goal of this analysis was to investigate whether the participants and *eXcomment* have a similar judgment regarding how much the comments describe TD situations.

We started by calculating the correlation coefficient among the scores, using two releases of our vocabulary. We found that in both releases there is not a strong correlation, but in second release the recalculation of the new scores improved the performance of the correlation coefficient in the analyzed projects. This means that the patterns responsible for comments considered false-positives affected the final scores, and their removal improved the correlation coefficient.

Once we analyzed the correlation coefficient, we calculated the difference among the scores. We identified many comments with low differences. This means that the participants and *eXcomment* had similar judgment regarding how these comments are important to detect a TD situation. This indicates that the final scores can be useful to highlight some comments that can really point out to TD items. We also found some comments with high differences. In order to investigate the divergence between participants and *eXcomment* in these cases, we performed a qualitative analysis in these comments.

In spite of the heuristics implemented in our approach to be considered important to detect semantics around the patterns, we found that there are semantic characteristics identified by humans that are not considered by our heuristics yet. Thus, new heuristics can be created to identify other combinations of patterns and improve the final scores calculation. Our tool and vocabulary need continuous improvement in order to cover situations that are not being addressed in the currently heuristics and to evolve the final scores calculation. We intend to analyze other projects in order to discover new patterns and heuristics to improve the contextualization power of our approach. We hypothesized that this will improve the precision of the comments identification reporting TD items through detection of patterns.

Another point we concluded is that the complexity of a comment can be an obstacle to participants and *eXcomment* interpret how much a comment describes a TD situation. In addition, we analyzed the participants' feedback form and the answers evidence that determining how much a comment describes a TD situation is not an easy task. New heuristics need to be created to interpret complex comments. Therefore, final scores can be used to sort comments and the detected patterns can support people to decide if a comment describes a TD situation or does not.

7.9.3 Identifying types of TD.

The results from this RQ show that on the one hand, the classification of our approach and participants were very similar to code and defect debt, showing that our vocabulary and themes are calibrated to identify these types through code comment analysis. On the other hand, some types did not have high precision considering the participants' classification and the classification done by *eXcomment*.

Regarding comments without types identified automatically by our approach, we found that although *eXcomment* did not detect any type of TD in some comments, the participants classified them in different types, mainly code debt. Besides, another aspect we analyzed was the contents on the comments with low precision. This anal-

ysis provides insights on the divergence between the classification of participants and *eXcomment*, which we present next.

We found that the relationships between some patterns, themes, and types of TD need to be reviewed. This relationship can be responsible for the *eXcomment* misclassification. Therefore, we intend to perform a deep qualitative analysis to examine such cases. Another factor is related to complex comments.

Besides, we hypothesize that the comment is more complex for participants to decide which type(s) the comments is(are) classified when: (i) a comment describes a situation that can be classified in more than one type of TD, (ii) the comment describes a TD situation that reports an intersection of concepts of different TD types, or (iii) the comment has much information about different issues.

We conjecture that some tips may support participants to make decisions on the TD identification process. For instance, the indication of the patterns, heuristics, and themes of TD that the comment may be associated can help developers think about the comment, analyzing deeply the location where it is and, maybe, the code related to it.

Once we identify the patterns in a comment, our heuristics are important to highlight the combination of patterns, facilitating the comments comprehension and in the final scores calculation. However, the heuristics are not prepared to classify a comment in a type of TD. Another fact is that some patterns are not related to a type of TD.

We also found that *eXcomment* is not able to identify types of TD in some comments because it did not identify some expressions describing a type of TD. Thus, new analyses need to be performed in order to discover new patterns and to relate them to type(s) of TD.

The last aspect we analyzed was the participants' feedback. Only one participant answered that he did not have difficult choosing types of TD related to comments. All of the others emphasized some difficulty choosing the types. In addition, we analyzed the notes in feedback form and we highlighted the main summaries which follow next (translated to English): (i) *I had some difficulties understanding and deciding on complex comments*; (ii) *I had the feeling that I needed to know the software project context better*; (iii) *I believe some tips on comments could help us to interpret and analyze complex comments*.

To conclude, evidences show that identifying TD through code comment analysis can be a hard task. However, our approach can be used by developers to identify automatically TD items in some cases, classifying them in some types of TD. In addition, even when that is not possible, it provides support to facilitate the analysis of comments by developers in order to detect TD items.

7.9.4 Identifying TD themes.

We found that our strategies worked well to identify automatically themes in the most of the comments analyzed in this experiment through comment analysis. We consider that *eXcomment* is calibrated to identify TD themes using a contextualized vocabulary.

The results show that in fact the automatic classification of themes is more precise than the automatic classification of types, considering the participants' judgment. We

conjecture that themes are more related to situation and causes of TD described in comments than types of TD. TD themes are a category of patterns which are related to a specific situation of TD, for example the theme "Open Task" is related to tasks that need to be done. Whereas the types have the intersection of concepts, which may cause doubts in the identification of the types of TD. Thus, themes identified automatically by *eXcomment* may help humans to identify TD and decide which type(s) of TD are related to TD items.

7.9.5 TD themes support the classification of TD items.

We found that themes are useful to support *eXcomment* and participants to classify correctly types of TD. On the other hand, we also observed that some themes do not support the identification of some types of TD. A possible reason for that is that some themes seem more related to a type of TD than others. So, some relationships theme-type need to be reviewed and calibrated in order to be useful for *eXcomment* to classify types of TD automatically.

The results confirm the previous findings that a comment can be related to more than one type of TD by *eXcomment* through relationship of themes-types. However, a type can be better evidenced than others by humans because a comment can better describe a theme than others.

Concerning the importance of TD themes to classify types of TD, we analyzed the participants' feedback form. Only two participants answered that TD themes are not useful to classify a type of a TD item through comment analysis. All of the other participants highlighted that TD themes are useful or quite useful for them to decide which type of TD a comment describes. This shows that the automatic identification of TD themes may support a tool and humans to identify and classify TD items.

The complete list of comments selected by *eXcomment* and analyzed in the *FindTD IV* is publicly available [30].

7.10 THREATS TO VALIDITY

We followed the checklist provided by [94] to discuss the relevant threats to this controlled experiment.

7.10.1 Construct Validity.

These threats concern the relationship between theory and observation. The first limitation is regarding the qualitative analysis performed in this study. We adopted a qualitative analysis to analyze the content on comments. In order to deal with subjective bias, the analysis was run by two TD experts. This strategy minimizes the subjective bias due to knowledge difference among the researchers.

Another threat involves the definition of the proposed vocabulary. It is possible that the set of patterns, scores, and heuristics used in the study are simply too many to be studied. An alternative would be to limit the studies to specific contexts and software domains.

Finally, to avoid social threats due to evaluation apprehension, students were not evaluated on their performance. Regarding the experiment planning, we made an effort to minimize communication among the subjects aiming to reduce the interference among their answers. Moreover, we clearly explained the process and introduced the support material – available during the experiment execution – to avoid the subjects' misguidance.

7.10.2 Internal Validity.

The first internal threat we have to consider is subject selection since we chose all participants through a convenience sample. We randomly divided the participants to analyze two different open source software projects in order to minimize this threat. Another threat is that participants might be affected negatively by boredom and tiredness. In order to mitigate this threat, we performed a pilot study to calibrate the time and amount of comments to be analyzed in each software.

Another threat to internal validity includes the judgment of how much comments describe a TD situation. To minimize this threat, we used a range of values on an ordinal scale from 1 to 10.

An internal threat is concerned to outdated comment. It is also important to deal with outdated code comments because they may wrongly indicate a TD item. Thus, developers may not remove comments when they pay (remove) a TD item. However, [75] and [65] examined this phenomenon and they concluded that code and comments consistently change and [69] concluded that source code and code comments have a high co-evolution, it found that 97% of the comment changes are consistent with source code.

Last but not least, the last internal threat is concerned to the way how the themes and TD types were analyzed by the participants. The training focused on how participants should select TD themes and types, reducing this threat. The participant feedback answers confirmed this mitigation because they reported their difficult to identify both themes and types. We also limited this bias in order to reduce the impact on results. For these reasons, each group member rated 65 comments in common with each other group member.

Finally, we performed an integrated qualitative with a quantitative analysis in order to increase the interpretation of the statistics results and reduce the internal threats.

7.10.3 External Validity.

Even though graduate student and software engineers analyzed data from real software, we cannot generalize the findings and their applicability to industrial practices. We chose a large and mature open source software projects to try mitigating this thread. All the experiment's participants have some professional experience in the software development process. It is an important aspect in mitigating the threat.

The last limitation covers the effort to carry out all studies because of the difficulty performing experiments in this area, for example, the threat relates to the generalization of the findings and their applicability to industrial practices. This threat is always present in experiments with students as participants.

7.10.4 Conclusion Validity.

Conclusion validity threats are mainly due to avoid the violation of assumptions. We limited our data analysis at considering statistic methods and qualitative analysis. The findings considered the data analysis of 22 participants in the experiment in two OSS projects. Our sample can be considered as appropriate to validate of the conclusions drawn from this study. Our sample consisted of graduate students and experienced software engineers. To avoid the violation of assumptions, we considered patterns, themes and types of TD identified in the FindTD III by different participants for data analysis. We also considered correlation test, normality test, (Shapiro-Wilk) and a non-parametric test (the Mann-Whitney test) and qualitative analysis performed by two TD expert researchers.

Chapter 8

This last chapter presents the conclusions and major contributions of the thesis. It also addresses our published papers and the recommendations for further research.

CONCLUSION AND FUTURE WORK

This thesis explored the TD identification area aiming at empirically investigating how code comments analysis can support the identification of different TD types, considering the developers' point of view. We propose a text mining-based approach to automatically recognize comments having a TD context. In our approach, we utilize an initial model, a contextualized vocabulary, and a tool (*eXcomment*), aiming to classify comments through searching for patterns that describe a TD situation.

We performed two mapping studies to help us understand the mining software repositories and code comment areas. We defined, evolved and evaluated our approach using strategies derived from the experimental software engineering area. We carried out a family of experiments composed of four studies (*FindTD I, FindTD II, FindTD III, and FindTD IV*). The empirical studies contributed to enlarging the knowledge about the TD phenomenon by presenting a new approach to identify and classify TD items. The findings from these empirical studies, which are complementary to each other, evidenced that the proposed model, vocabulary, heuristics, and search strategies are an important contribution because they can be used to develop and improve techniques with a focus on the identification and management of TD items.

8.1 REVIEW OF THE MAIN FINDINGS OF THIS THESIS

The main contributions of this thesis are related to the knowledge on how comments could be used to automatically identify and classify TD items through code comment analysis and to the family of experiments which we carried out. To ease replication of the experiments presented in this thesis, all the data used in the studies are publicly available (releases of the vocabulary, comments extracted in each study, the participants' answers, and the analysis). Following, we present the main findings of each study.

FindTD I was proposed to evaluate the proposed model and its vocabulary through an exploratory study on two large open sources projects. The study had as the primary

goal to characterize the feasibility of the approach to support TD identification through source code comments analysis. The model provided a vocabulary based on a structure that systematically allows combining terms creating a large set of patterns on TD. The main result of this step indicates that the model provides a vocabulary that can be used to detect TD items.

To make more consistent the initial findings, we have performed a controlled experiment, *FindTD II*. This experiment had as primary objective to extend the first study analyzing the use of the model and the contextualized vocabulary by considering the overall accuracy when classifying candidate comments and factors that influence the analysis of the comments to support the identification of TD regarding accuracy. The results indicate that:

1. In spite of non-native English speakers are frequently unaware of the most specific terms used in general English, the patterns of the vocabulary were useful to support the understanding of comments considering the specific scenarios of TD;

2. Comments selected by our approach may be understood by either an experienced or non-experienced observer. This reinforces the idea that the TD concept aids discussion by providing a familiar framework and vocabulary that may be easily understood;

3. Many analyzed comments were considered good indicators of TD by participants of the studies. In general, the results provide evidence and motivation for continuing to explore code comments in the context of TD identification process.

In order to carry out a broad analysis of the contextualized vocabulary, we performed the third study, another controlled experiment, *FindTD III*. In this experiment, we used a qualitative method for identifying the most significant patterns, themes related to TD context, and the relationship between patterns and different TD types. By doing this, the vocabulary was improved considering the classification of the most important patterns to identify TD. The main results of this study indicate that:

1. Over half of the patterns identified in FindTD II are decisive or very decisive to identify TD;

2. Some patterns are considered more contextualized and decisive than others to help observers on identifying of comments that report a TD situation. We also observed that other patterns only make sense when they are combined with other ones;

3. We identified 37 different TD themes related to patterns (which describe a TD context) and to TD types. We used TD themes to detect which TD types exist in the software projects explored in the two performed experiments;

4. The model and the vocabulary evolved over the three releases. The evolution of model and increase in the vocabulary can improve the level of contextualized information to support the comments filtering and the TD identification.

Following our early works, we set out to carry out the last experiment (*FindTD IV*). The primary goal of this study was to evaluate our complete approach to identify and classify TD items through code comment analysis automatically. Also, it aimed at broadening the investigation (performed by other studies) on the comments selected by our strategies, evaluating whether they describe a TD situation. This evidence also allowed us to investigate the contribution of the vocabulary to support the TD identification process. The main findings of this study are:

1. We found some patterns that were responsible for comments considered false-positives and they are not useful to identify TD items through code comment analysis. These patterns affected the final scores, and their removal improved the correlation coefficient;

2. The final scores calculated by the *eXcomment* can be useful to highlight some comments that can point out to TD items;

3. Our tool and vocabulary need continuous improvement in order to cover situations that are not being addressed in the current heuristics and to evolve the final scores calculation;

4. Another point we concluded is that the complexity of a comment can be an obstacle to participants and eXcomment interpret how much a comment describes a TD situation;

5. The heuristics implemented in *eXcomment* are important to highlight the combination of patterns, facilitating the comments comprehension and in the final scores calculation;

6. We find that our approach was more efficient for identification of code and defect debts through code comment analysis;

7. Some evidence shows that identifying TD items through code comment analysis can be a hard task. However, our approach can be used by developers to determine TD items automatically in some cases, classifying them in some types of TD. Also, even when that is not possible, it provides support to facilitate the analysis of comments by developers in order to detect TD items;

8. We found that our strategies worked well to identify themes automatically in the most of the comments analyzed in this experiment. We consider that *eXcomment* is calibrated to detect TD themes using a contextualized vocabulary;

9. We found that themes are useful to support *eXcomment* and participants to classify types of TD correctly.

These findings allowed us to conclude that code comments provide a perspective which permits to consider the developer's point of view to complement the quantitative analyses (from code metrics extraction) in the TD identification process. Code metrics-based tools

and our approach may contribute to each other to make the TD identification process more efficient. Different from code metrics-based approaches, comments analysis allows to achieve aspects behind the source code, access the developers' point of view on what is considered as a TD item or not. In this sense, code comments are a valuable software artifact which stores richness of semantic information written in natural language and may describe different TD situations.

8.2 PUBLISHED PAPERS

We present some of these results and some contributions of this thesis to the scientific community to obtain feedback and improve the findings that we have discussed. We have also published papers in international conferences, presenting achievements related to the required courses of the doctoral program, works related to the global area of this thesis (Technical Debt), and the researches linked with the central topic of this thesis (Identifying TD through Code Comment Analysis). Figure 8.1 summarizes the publications, but for the sake of simplicity, we did not detail the papers that are beyond the main scope of this thesis. Moreover, some discussions and contributions have not yet been published.

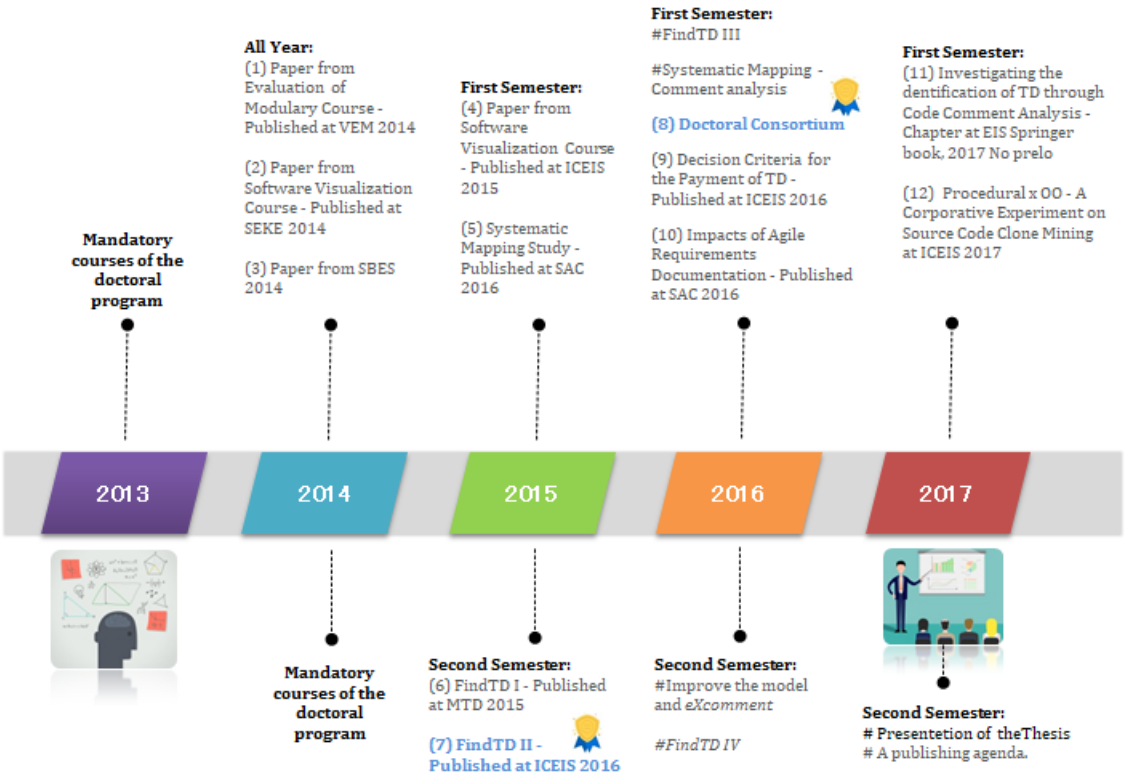


Figure 8.1 Timeline of the research.

- *A Systematic Mapping Study on Mining Software Repositories* (5). We published this at the 31th ACM/SIGAPP Symposium On Applied Computing (ACM-SAC

- 2016). We performed a systematic mapping study in order to understand the MSR area and to identify its current targets and gaps, regarding mainly source codes and comments analysis. The results of this work showed lacks in this area and provided the first evidence supporting the begin of this research. Moreover, we identified some important studies which focus on the usage of comments for software comprehension, and some techniques and tools used to extract and analyze comments. We present this paper in the Appendix A of this thesis.

- *A Contextualized Vocabulary Model for identifying technical debt on code comments* (6) . This paper was presented at the 7th International Workshop on Managing Technical Debt (MTD - 2015). It discusses the results of our first experimental study, showing the CVM-TD, a model to support the identification of technical debt through code comment analysis. This model provided an initial contextualized vocabulary which allows us filter candidate comments from source code. The paper also presents the analysis of the *FindTD I* and the *eXcomment*. The study evolved the vocabulary (second release). It is important to highlight that this study was the basis to plan and perform the rest of experiments we have performed in this thesis.

- *Investigating the Use of a Contextualized Vocabulary in the Identification of Technical Debt: A Controlled Experiment* (7). This work was published at the 18th International Conference on Enterprise Information System (ICEIS - 2016). It presents the *FindTDII* (a controlled experiment) and the results allowed us to calibrate the vocabulary and motivate us to continue exploring code comments in the context of the TD identification process to improve CVM-TD and the *eXcomment*. It provided the third release of our contextualized vocabulary. This work presents part of this research. We underline that the work won the best student paper award in the conference.

- *Identifying Technical Debt through Code Comment Analysis* (8). We published a doctoral proposal at the Doctoral Consortium on Enterprise Information Systems (DCEIS - 2016). This work presents our research plan and methodology that was applied in this thesis. We highlight that the work also won the best Ph.D. project award.

- *Investigating the Identification of Technical Debt through Code Comment Analysis* (11). This work was published as a chapter of the book Enterprise Information Systems (Springer) - 2017. It extends the findings and analysis of the *FindTDII*.

8.3 FUTURE WORK

The approach that we have explored in this thesis brought us some possibilities about additional discussions that can be addressed in future works. We present some possible future work agenda following the gaps identified in our experiments:

1. The research community can perform a deep analysis on some cases that warrant further examination to identify other patterns that can impact on detection of TD items.

2. To improve the proposed heuristics and to define other heuristics to detect TD items and reduce the number of false positives;

3. To enrich our family of controlled experiments. To do that, an exploratory study on an industrial system can be performed.

4. To compare our approach to the ones proposed by Shihab et al. in [2], [11], and [99]. Another study, in progress, is to compare our approaches to tools that use metrics extracted from the source code to identify TD items, in order to confirm or to contradict our findings;

5. We also intend to implement some visual paradigms to visualize the extracted data to facilitate the comprehension of the discovered patterns and transform them into knowledge (Interpretation phase);

6. Our approach can be used by the community to analyze other datasets from other open software projects domains and projects developed in different programming languages;

7. We plan to interview some contributors in open source or industrial projects to gather information about the how code comments can be useful for the TD identification.

8. Finally, machine learning techniques can be applied, using our contextualized vocabulary, to improve the accuracy of our approach.

Appendix

A

A SYSTEMATIC MAPPING STUDY ON MINING SOFTWARE REPOSITORIES

In this appendix, we present the complete systematic mapping study on mining software repositories.

In order to understand the MSR area and to identify its current targets, shortcomings, and gaps, regarding mainly source codes and comments analysis, we performed a systematic mapping study over works published at five years (from 2010 to 2014) of Working Conference on Mining Software Repositories (MSRConf). We chose this fresh period in order to explore recent studies on MSR area. This allowed us to investigate how researches are being conducted in this field and to address our research questions.

We chose this conference because the MSRConf is the main conference in the area disseminating results of works on new processes, methods, techniques, tools, and ideas in this field [42]. Despite we know there are other venues about the topic (e.g. Empirical Software Engineering and Measurement, Empirical Software Engineering Journal), we focused on MSRConf that we believe is representative for this work.

A.1 SYSTEMATIC MAPPING METHOD

Some researchers have been working to provide stable methods for systematic literature review process [100–103]. In this study, we applied the approach defined by Petersen et al. [101] and Group [104] in order to define the procedure of our mapping study.

We chose to conduct a mapping study because it allows analyzing the primary studies aiming to answer our research questions in a broader way and to collect evidence for directing our future research. In addition, both data extraction and analysis are largely concerned with classification of the available studies. The steps of the performed mapping study are detailed in the following subsections.

A.2 RESEARCH QUESTIONS

The main purpose of the mapping study is to identify and characterize the set of recent primary studies published at the MSRConf in order to identify the current picture of the

MSR area. Thus, the main research question, which guides our study, is: *"What are the current goals addressed by recent approaches in MSR area?"*.

To answer this question, we derived other three specific questions, described at the following:

RQ1. *Which are the main purposes, focus, and object of analysis of the identified approaches?*

This question aims to identify the goal of MSR approaches regarding facets based on classification proposed by Basili and Lanubile [23]:

- **Purpose**: defines the main intention of study. The purpose of a study can be classified in characterizing, comprehension, evaluation, controlling, improving or prediction something [23]. Besides these categories, we also considered localization, classification, and identification, because they are frequently explored in the MSR field [64];
- **Focus**: is the main attribute of interest between purpose and object of analysis;
- **Object of analysis**: is the entity of interest in the analysis;
- **Point of view**: describes the perspective of a person needing the information;
- **Context**: describes the environment in which the analysis is performed.

Purpose, focus, and object of analysis are generally used together in MSR analysis. For example, to predict (purpose) defects (focus) analyzing bug reports (object of analysis) [57], and comprehending (purpose) power consumption (focus) analyzing mobile apps (object of analysis) [57]. In order to capture the goal of the studies, we focused on the first three facets for data extraction in our mapping study. Most papers did not explicitly mention what is the target audience for the approaches they propose (point of view) and the environment in which the analysis is performed (context).

To perform the classification, we created a taxonomy based on proposed classifications [64][105] and grouped the different terms used by researchers for presenting purpose, focus, and object of analysis into categories. The classification scheme will be presented Section A.5, when we show the gathered data in the mapping study. The taxonomy used in our study is publicly available [63].

RQ2. *Which are the data source types supported by mining software repositories and how are they evolving?*

This question aims to know which software mining infrastructures are used for MSR investigations and how they are evolving over time. According to [55], software mining infrastructure may be subdivided in three major categories: (i) historical repositories, such as source configuration management (SCM), bug tracking systems (BTS), and archived communications recording part of information about the evolution and progress of a project; (ii) run-time repositories, such as deployment logs, containing information about the execution and the usage of an application at a single or multiple deployment sites; and (iii) code repositories, such as Sourceforge.net, containing the source code of various applications developed by several developers.

Normally, source code repositories are used to access information about a specific software version (snapshots), or the software evolution history [46]. SCM systems are primarily used to register and maintain changes to source code artifacts. Moreover, SCM metadata has important information to support software comprehension. Examples of such information are: commit messages, commit date-time, commit author and changed data. BTS records maintenance changes and keeps track of the life cycle of bugs. This type of repository also may contain textual description of the bug, its location, and developers involved with the bug [58]. Archived communications store several types of discussions and requests for help among software team members. Examples of such repositories are mailing lists, questions and answers forums, and microblogging.

These repositories possess different information content, storage format, and vary in the way researchers mine their data [64]. To better analyze the data source type, we subdivided the repositories into two subcategories: structured and unstructured (Level 1). Level 2 (sublevel of Level 1) denotes the types of structured and unstructured repositories. It was subdivided in: (i) structured repositories: binary code, ITS, BTS, SCM metadata and source code, and (ii) unstructured repositories: mailing list, document, forum and log. In Level 3, we summarized the terms that characterize repository types of Level 2, such as "jar file" to represent "binary code" and "iBugs dataset" to represent "BTS".

RQ3. *Which are the conducted empirical evaluations?*

This question aims to identify whether the proposed approach has been evaluated through empirical methods, and if so, which method was used. We considered that a study has an empirical evaluation if it brings at least one section with some discussion dedicated to this topic. The empirical method is classified in one of the following types [17][106], depending on the purpose of the evaluation and how researchers described their evaluated methods:

Survey: is an empirical strategy to gather data from a population sample and to achieve an understanding of that population. The data collection process is achieved through questionnaires. The facts, opinions, or beliefs of the people are polled to determine how the population reacts on a particular topic [107];

Case study: is a research methodology that studies a phenomena in its natural context [108]. We can find case studies that range from very ambitious and well-organized studies to small toy examples that claim to be case studies. In this sense, we respect the authors' assumptions;

Controlled experiment: provides a formal, rigorous, and controlled investigation in which an intervention is introduced to observe its effects. Controlled experiments are normally used for identifying cause-effect relationships and the factors that may influence the outcome should also be controlled [109];

Feasibility Study: in this type of study, authors use the proposed approach over a software as a feasibility study and it only presents a proof of concept [110];

Exploratory Study: is conduced to analyze a problem that has not been defined. It focuses on exploring a specific context and does not intend to discuss final and conclusive solutions, but helps researchers to have a better understanding of the problem and generating ideas and hypotheses for new research [108];

Comparison with other tools: The authors performed a comparison based on

checklist among their approach and another related tool [105].

A.3 SEARCH STRATEGY

We used a manual search to retrieve the studies for this mapping study. We considered full papers published at MSRConf from 2010 to 2014. We selected this conference because it is the main venue of this area. To ensure that results of the study selection are objective, we defined inclusion and exclusion criteria.

Inclusion criteria. The study need to be a full paper published at MSRConf on the selected period. Full papers present mature approaches, and likely more established studies. The study needs to explore a theory, a practice, or an approach related to the MSR process and software engineering domain.

Exclusion criteria: We excluded studies that do not address mining software repository and software engineering domain (e.g. [111]). Surveys and secondary experimental studies were removed, since they report approaches from others. Short papers, challenges, and showcases were also excluded (e.g. [42], [56]).

The inclusion and exclusion criteria over the five years of MSRConf resulted in 107 papers to be further analyzed and classified. The number of papers per year is as follows: 2010 – 15; 2011 – 19; 2012 – 17; 2013 and 2014 – 28 each.

A.4 DATA EXTRACTION PROCESS

Two researchers developed a taxonomy used in the data classification process. The first author performed a data extraction pilot with students from an experimental software engineering (ESE) discipline at a master and PhD course. In that moment, we discussed the application of the taxonomy in order to make clear the meaning of data.

After the pilot, two researchers have carefully read each paper and, in the next step, we have done a consensus process for each paper. If after that the researchers disagreed in a classification or extracted information, a third opinion was used to reach agreement.

The extracted data were recorded on a spreadsheet that is publicly available [63]. We analyzed the extracted data quantitatively in an effort to answer our research questions.

A.5 RESULTS AND DISCUSSION

In order to analyze the following research questions, we considered that one study can be classified in more than one classification. For instance, "comprehension" and "prediction" for purpose, "defects" and "changes" for focus, and "source code" and "commit data" for object of analysis.

RQ1. *Which are the main purposes, focus, and object of analysis of the identified approaches?*

Purpose: As explained in Section 3.2, we used [23] to classify purpose – approach also followed by the papers [64][105]. Figure A.1 shows the purpose category we found in our mapping study and the number of studies per each category. The category "comprehension" has the highest number of studies (58 – 47.9%). From it, we can see that: 58 - 47.9% of the studies present at least one approach that aims to understand some type

of software artifacts; 20 - 16.5% were classified as “prediction”; 18 - 14.9% as “identification”, and 12 - 9.9% as “evaluation”. These four purposes represent 89.2% of studies we analyzed in this mapping. Controversially, the other four purposes represent only 10.8% of all studies. We did not identify any study associated with “controlling” purpose.

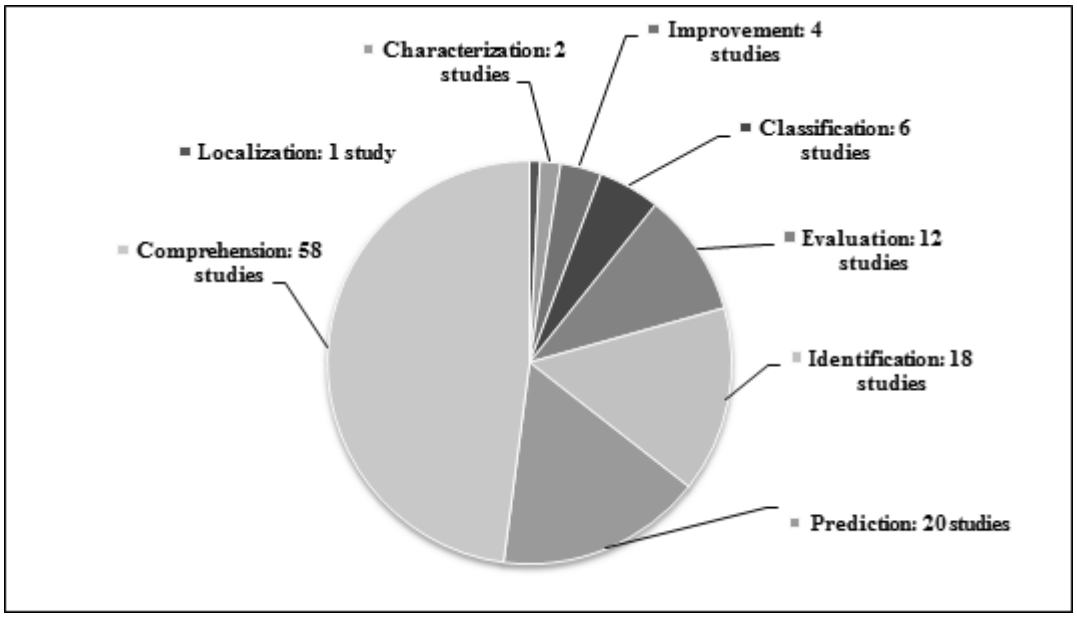


Figure A.1 Purpose categories.

Focus: Figure A.2 shows the focus category we found in our mapping study and the number of studies per each category. In addition, we can note that focus facets “defect” with 39 – 31.2% (also related to bug topic), “changes” with 10 – 8% (also related to change), and “software evolution” with 8 – 6.4% (also related to bug evolution) represent almost half of studies we analyzed. They have been the most explored focus by the analyzed studies. The results are aligned with Demeyer et al. [46]. They revealed that MSR area has a heavy bias towards change and software evolution. Terms like “bug”, “change”, and “evolution” appear nearly every year and they were classified as one of the most popular research topics and as trend of MSR field.

Object of analysis: Figure A.3 shows the distribution of studies per object of analysis type. “code” with 41 - 28.5%, “commit data” with 29 - 20.1%”, “bug reports” with 26 - 18.1%, “comments” with 13 - 9.0%, “microblogging” with 8 - 5.6%, and “e-mail” with 6 - 4.2% are the six most significant objects of analysis. They represent 85.4% of all studies. In spite of we have only 19 studies that mined unstructured repositories, the numbers of unstructured entities of interest in the analysis are significantly relevant. There is at least one study that analyzes microblogging in each year with the exception of 2010.

Here, it is important to highlight that the popularization of questions and answers systems like Stackoverflow , used by software team members, contributed for propagation of “microblogging” as object of analysis. They store a rich sort of discussions

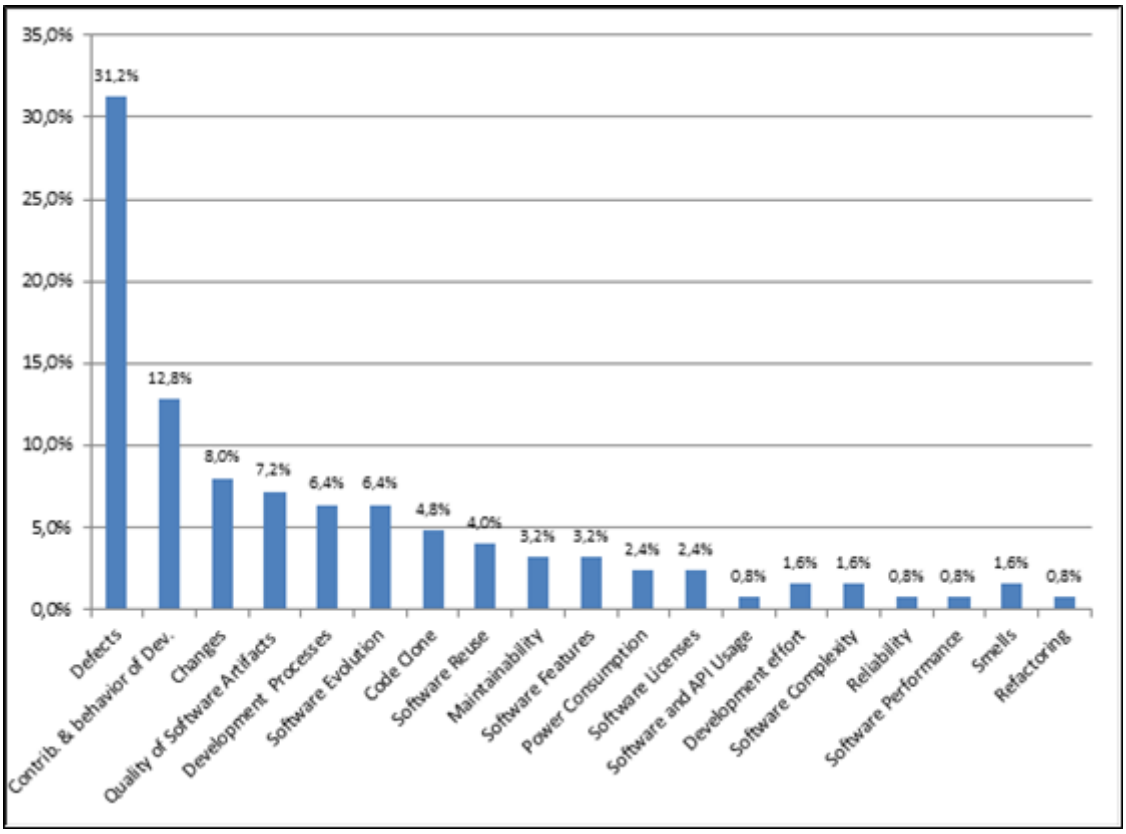


Figure A.2 Focus categories.

within the community where software project participants try to help each other. Stackoverflow is the main example of microblogging category with 75% of the studies (e.g. [112][113][114][115][116][51]). Text mining and natural language processing is being commonly used to mine text field in structured object, such as bug description in bug report, commit comments in commit data and code comments in source code (e.g. [117][118][119][72][71][120]). It is also important notice that none work that had unstructured repositories as object of analysis focused on TD identification.

Purpose x focus x object of analysis: In order to perform a deeper analysis, we created a bubble chart (Figure A.4) to analyze focus, purpose and focus, and object of analysis together. "Comprehension" and "defects" are respectively the most used purpose and focus in MSR field. At the right part of Figure A.4, a bubble denotes one or more studies that include a purpose dealing with a specific focus type. Note that "comprehension" was combined with nearly every analyzed focus facets in this mapping (e.g. comprehension of code clone and comprehension of changes). Despite "prediction" has been assigned as the second most used purpose, most of these studies have been focused in defects and changes.

On the other hand, no work purposes comprehension of refactoring and software performance. Another interesting point is that "defects" is the only category that has been explored by all analyzed purposes. Those studies have explored nearly every object of

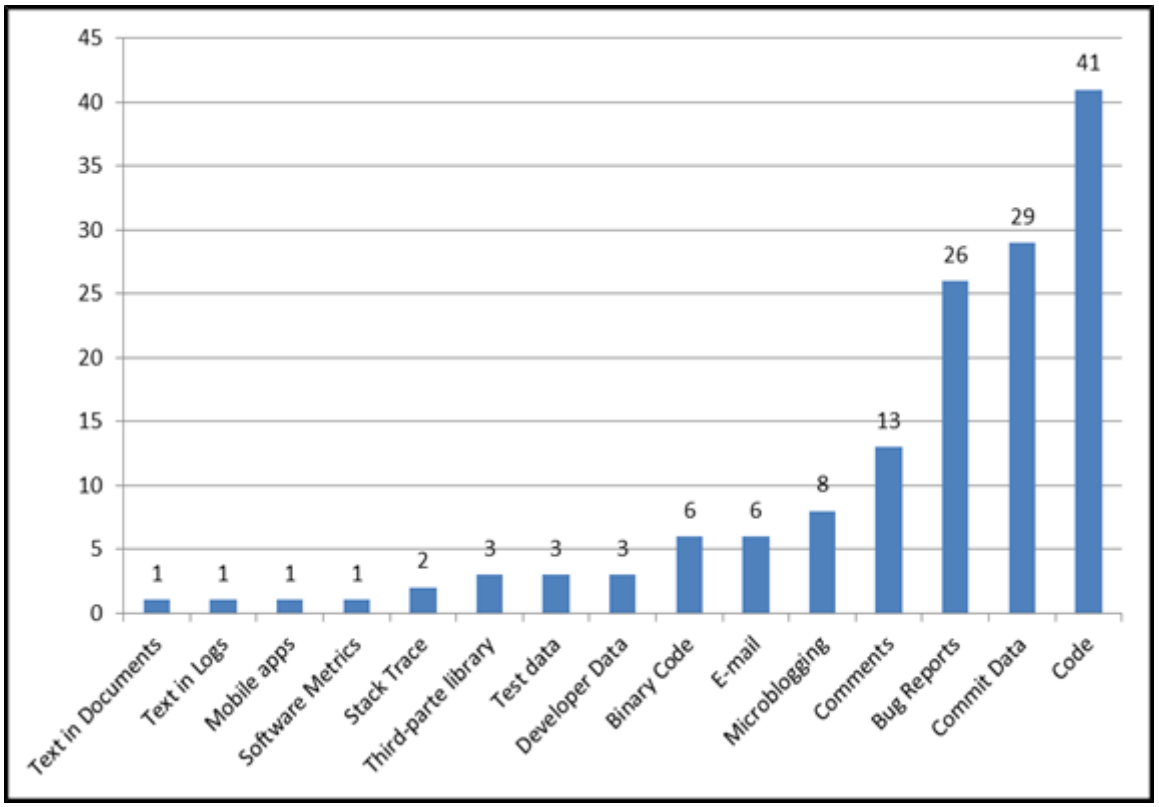


Figure A.3 Studies per object of analysis.

analysis to achieve their goals. In addition, researchers have heavily used "comprehension", "prediction", "identification", and "evaluation" combined with "defects" in their approaches.

The left-top bubbles in Figure A.4 show that "defects" is the most common focus of the analyzed studies. To do so, researchers have heavily used "code", "commit data", "bug reports", and "comments". Studies with focus in "changes" and "code clones" have heavily used "code" as main object of analysis. Many studies have focused on analysis of metrics extracted from "source code" and "bug report" to "understand defects". Although this combination has shown usefulness to support the understanding of defects, other objects of analysis have been studied in order to achieve the same purpose, such as "comments", "emails", and "microblogging". This may indicate that other software engineering artifacts are coming to be used to understand quality issues in software projects.

The usage of comments to identify smells, part of code that needs to be refactored, or quality of software artifacts have not been explored yet. Other gap is that no studies correlate "comprehension" with "refactoring", and "comprehension" with "software performance". We have only three "power consumption comprehension" studies in the mapping. They analyze "mobile apps", "microblogging", and "commit data". We can also see that other approaches have remained unexplored up to this point, for example: "localization of refactoring", "improvement of performance", and "localization of smells".

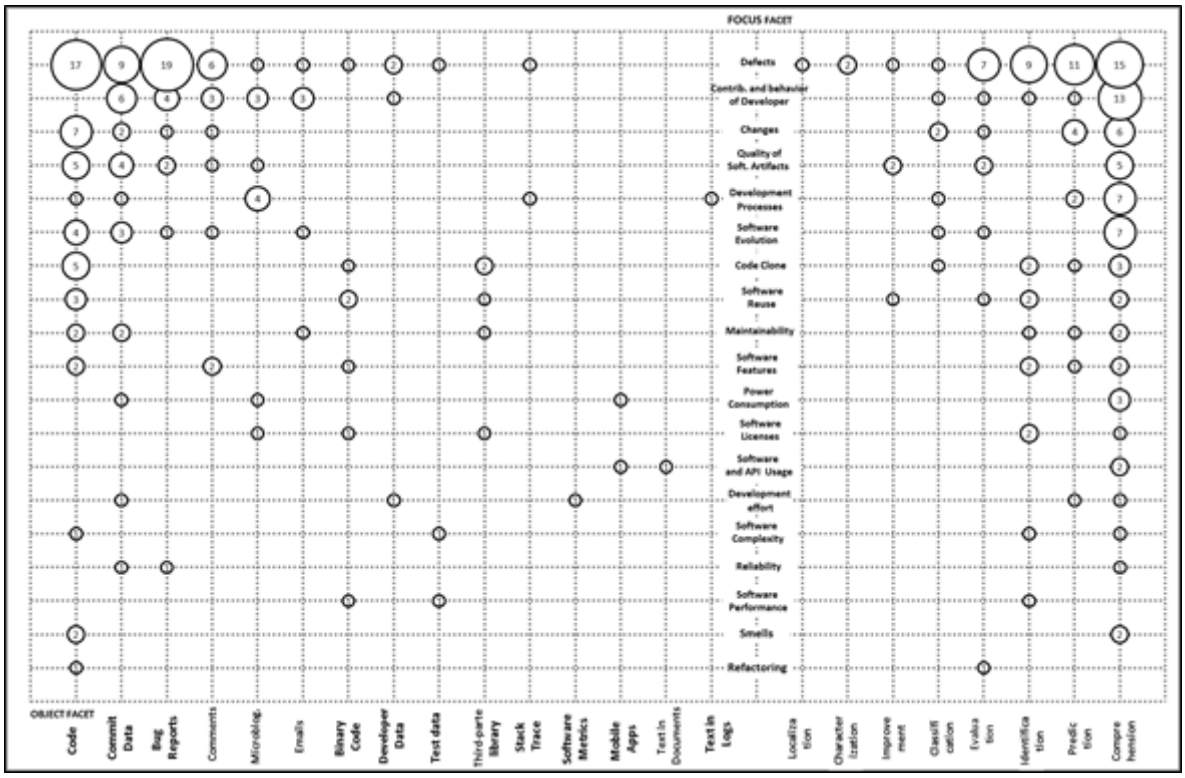


Figure A.4 Focus x object of analysis x purpose.

RQ2. *Which are the data source types supported by mining software repositories and how are they evolving?*

Recent researches indicate "source code", "SCM metadata", "BTS", and "archived communications" as the main data sources for MSR investigations [46][42][64][105]. Our results reinforce the indication of these studies. In addition, we found some unfamiliar data source such as "binary code", "issue tracking system (ITS)", "documents", and "logs".

Figure A.5 presents the results for data source types extraction. As explained in Section A.1, it is organized in three levels: (i) Level 1 represents the software mining infrastructure (structured and unstructured repositories); (ii) Level 2 denotes the main categories for structured and unstructured repositories ("binary code", "ITS", "BTS", "SCM metadata", "source code", "mailing list", "documents", "forum", and "log"), (iii) Level 3 summarizes terms we have found that characterize the types of repository. Again, one study can use more than one data source type.

The numbers inside the parentheses represent the total of studies per each category. We can see, for example, that "source code" with 45 studies – 31.5%, "SCM" with 37 studies – 25.9%, and "BTS" with 31 studies – 21.5% are the most used software repositories. Similarly, structured repositories are more common with 127 studies. A possible reason may be that mining techniques are more challenging over unstructured data or that those repositories do not provide rich information to achieve MSR goals.

Other data sources we found are: (i) 'iBugs dataset" with 2 studies, (ii) "Microblog-

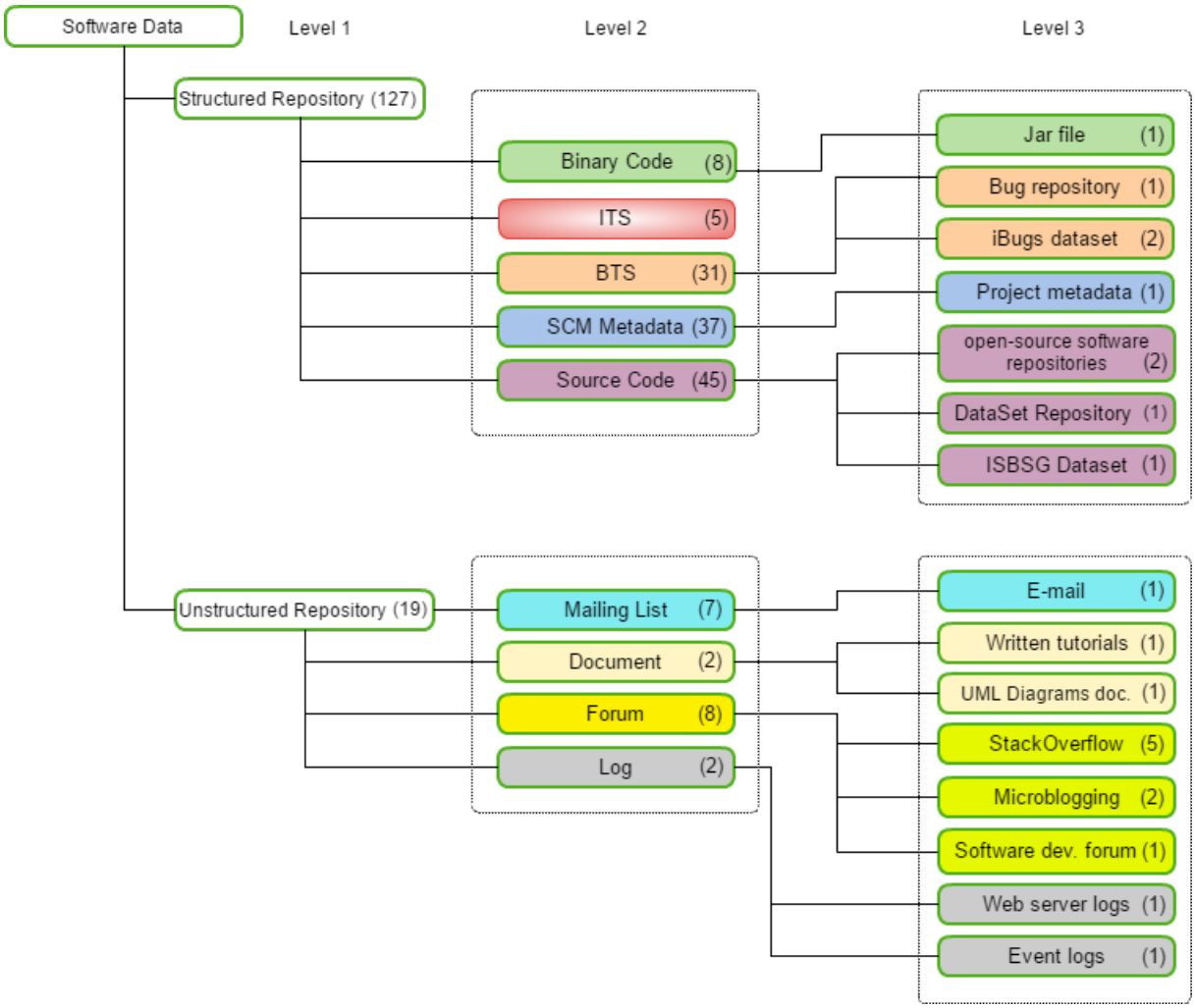
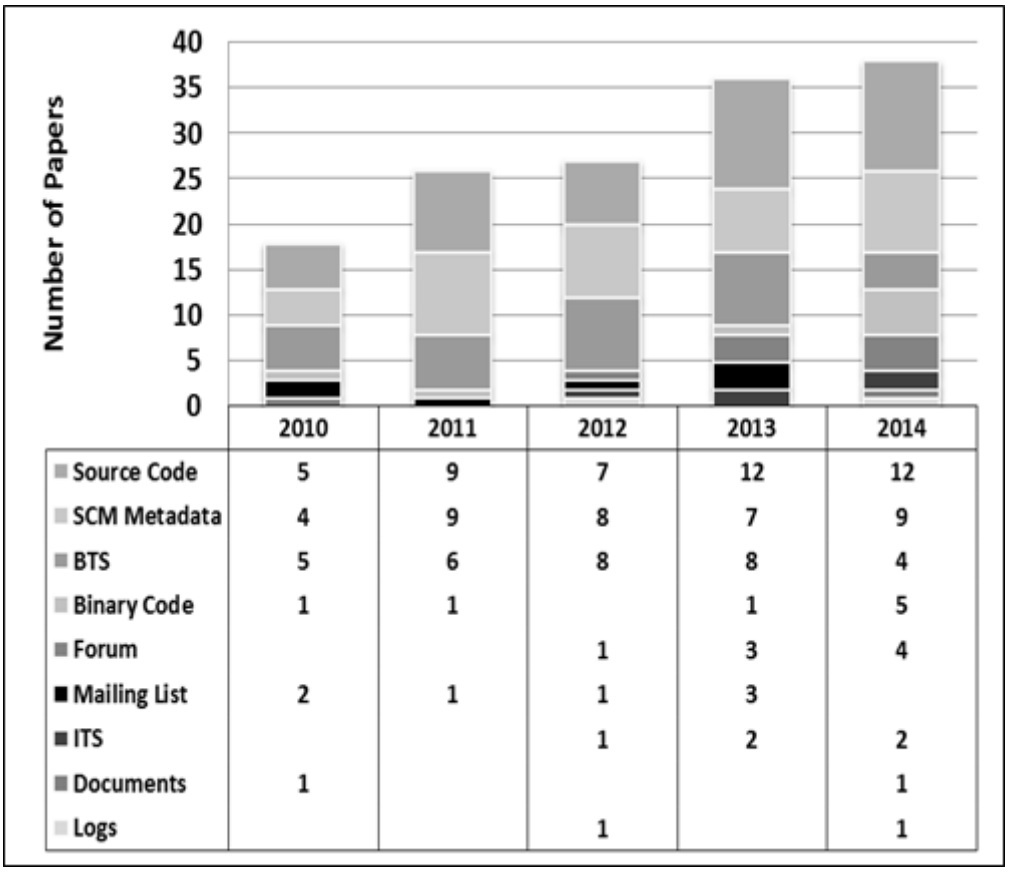


Figure A.5 Data source types.

ging" with 2 studies, and (iii) "Stackoverflow" with 5 studies.

Figure A.6 shows data sources distribution over the investigated years. We highlight two outcomes: (i) "source code", "SCM", and "BTS repositories" have been continuously used over the years; and (ii) the number of approaches using "archived communications" data source ("mailing list" and "forum") increased, reaching 12 studies in the last three years.

We defined a taxonomy that divided the data source types in structured and unstructured repositories. From this analysis, we might identify the main data source types in each category. Structured is more explored than unstructured repositories, but the number of approaches using unstructured data source is increasing in the last three years. Other software engineering artifacts are coming to be used, such as comments, emails, and microblogging. They have been analyzed alone or together with metrics extracted from structured repository to understand quality issues in software projects. Another point is that comments may describe quality problem in the development software, thus



	2010	2011	2012	2013	2014
Source Code	5	9	7	12	12
SCM Metadata	4	9	8	7	9
BTS	5	6	8	8	4
Binary Code	1	1		1	5
Forum			1	3	4
Mailing List	2	1	1	3	
ITS			1	2	2
Documents	1				1
Logs			1		1

Figure A.6 Data source by year.

code comment analysis a gap which can be explored in order to design and evaluate new methods and tools to support TD identification.

Finally, we identified several approaches combining more than one data source to achieve their goal. Figure A.7 shows that 31.78% of the studies make joint use of different software repositories. The most recurrent combinations are "SCM metadata + BTS" and "SCM metadata + source code" representing 16.82% of all papers. An important note is that "SCM metadata" is present in both cases and in five more, totaling 63.64% of all combinations. Therefore, our analysis pointed out that "SCM metadata" performs an important role in MSR area. These results are aligned to results presented by Novais et. al. [105].

RQ3. *Which are the conducted empirical evaluations?*

The studies analyzed in this mapping have conducted some types of empirical evaluation. Only one study did not conduct any evaluation [121].

We classified the studies according to six different evaluation methods. Figure A.8 shows the distribution of each of them. More than half of the published studies had been evaluated through an "exploratory study". They represent about 58 studies – 51.8%. "Case study" represents 31 studies – 27.7% and "Controlled Experiment" with 8 studies - 7.1%. The rest of studies were evaluated through "comparison with other tools" and

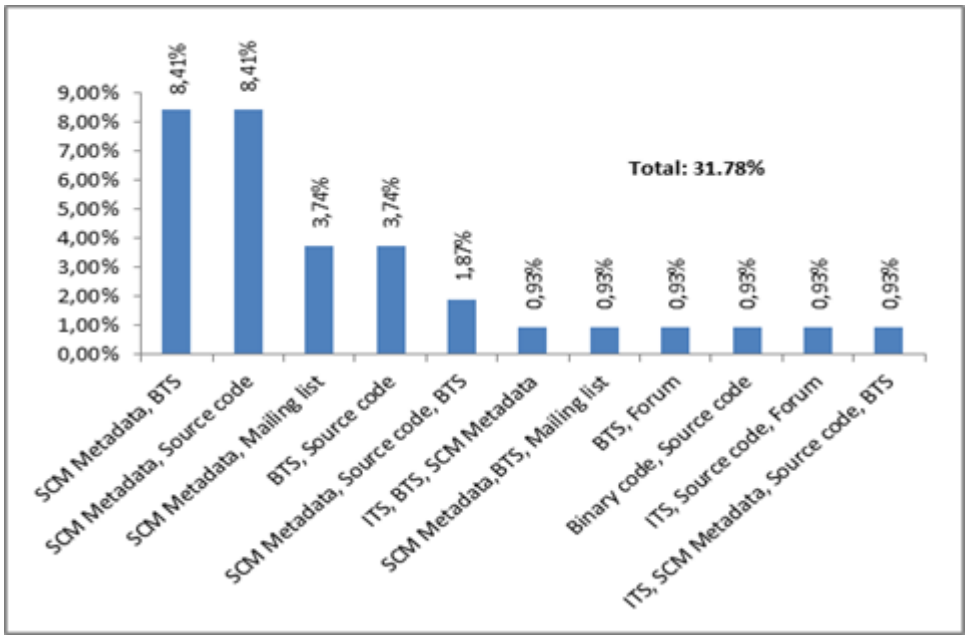


Figure A.7 Studies per data source types.

"feasibility study" with 5 studies - 4.5% each one, and "survey" with 4 studies – 3.6%.

Figure A.9 summarizes evaluation methods performed per purpose. We can observe that exploratory study and case study are the main research methods used for evaluating "comprehension", "prediction", and "identification" tasks. An interesting point is that "evaluation" and "prediction" were the only two categories where researchers have used all empirical methods to evaluate their approaches. Finally, we also can see that "survey" and "controlled experiments" have clearly been not much explored. In this sense, a deep analysis on these evaluation methods may be carried out, considering these types of studies.

A.6 SUMMARY OF FINDINGS AND INSIGHTS

In this study, we performed a systematic mapping study on studies published at five years of MSRConf. We have extracted and analyzed data from 107 papers. Our findings show some trends on current use of purpose, focus, object of analysis, source data, and evaluation methods at MSRConf.

A large set of purposes and focus were used to investigate MSR artifacts, but "comprehension of defects" and "code" was the most used purpose, focus, and object of analysis in this field (RQ1). We defined a taxonomy that divided the data source types in structured and unstructured repositories. From this analysis, we might identify the main data source types in each category. Structured is more explored than unstructured repositories, but the number of approaches using unstructured data source is increasing in the last three years. Other software engineering artifacts are coming to be used, such as comments, emails, and microblogging. They have been analyzed alone or together with metrics extracted from structured repository to understand quality issues in software

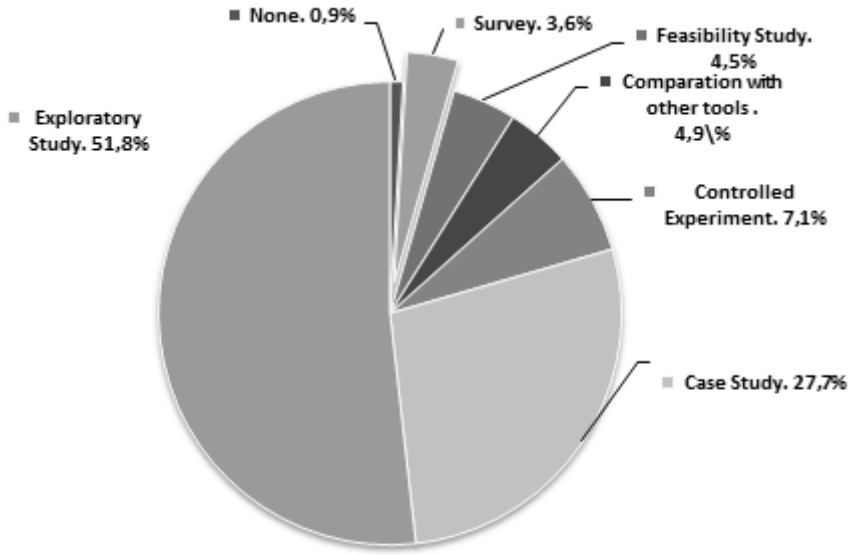


Figure A.8 Evaluation methods.

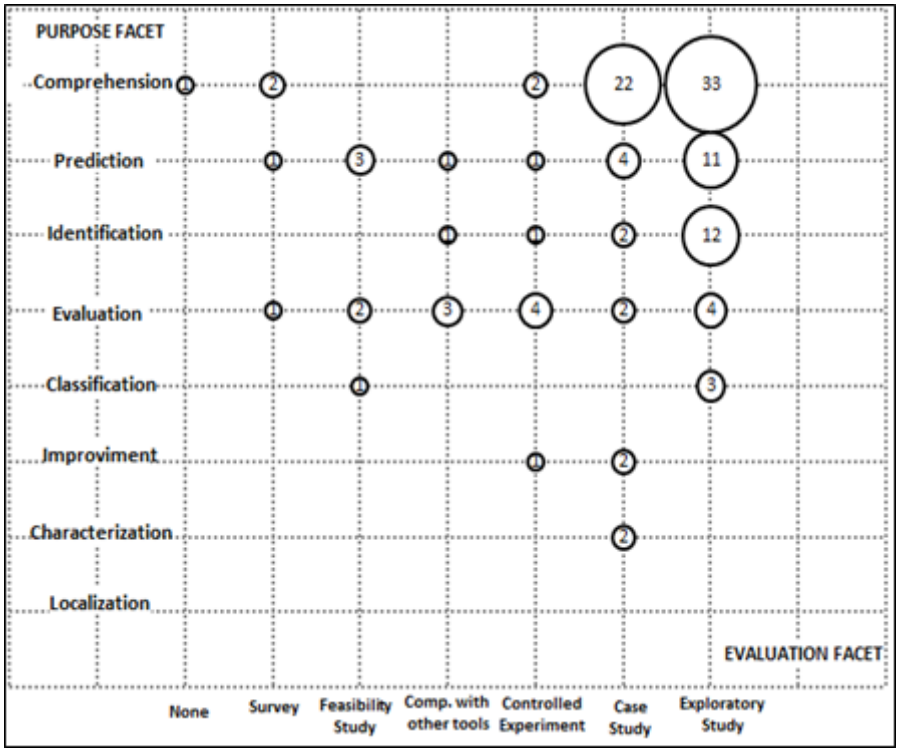


Figure A.9 Evaluation methods.

projects (RQ2).

Considering evaluation methods, we identified that almost all studies have performed some type of empirical evaluation and a low number of works has conducted "controlled experiments" and "surveys" (RQ3). Furthermore, other contribution of our study is that we make available the dataset of our data extraction and our classification [63].

This study shows us some gaps identified in the MSR area. This work is interested in the gap between code comment analysis and TD identification. This encouraged us to start investigating characteristics of software that may indicate quality problem in the software development, analyzing the code comments domain in order to design and evaluate new methods and tools to support TD identification

A.7 THREATS TO VALIDITY

Selection Bias. At the start of the process, we applied the inclusion and exclusion criteria based on our judgment and the studies were included or excluded into mapping study. This means that some studies could have been categorized incorrectly. With the intention of mitigate this threat, we discussed the study protocol among the researchers to guarantee a common understanding. After that, two researchers read each paper and a third researcher was involved in the discussion when needed.

Construct validity. The research questions defined in this study may not completely cover the MSR area in spite of questions have addressed the main aspect of MSR field, as discussed by [35][16]. We defined facets based in Basili and Lanubile [23] in order to try answering the research questions.

Data extraction. Bias or problem of extraction can affect the classifications of facets and the analysis of selected studies. In order to reduce this bias, we discussed deeply the definitions of data items, the initial taxonomy, and our extraction form. After that, we performed a pilot data extraction and the doubts were discussed and solved in order to make clear the meaning of data items to other researchers. Finally, two researchers conducted in parallel and independently the data extraction process and mitigated occurred conflicts in a consensus meeting. If the researchers disagree about a classification or extracted information, we used a third opinion to resolve differences and reach agreement to make sure that the extracted data are valid and clear for further analysis.

External validity. We performed a systematic mapping study over studies published over five years of MSRConf. This point implies that we may have missed some relevant studies. Thus we cannot generalize our conclusions for whole MSRConf. However, our outcomes allow us to draw insights to guide further investigations

Appendix

B

A SYSTEMATIC MAPPING STUDY ON MINING CODE COMMENTS

This appendix presents a systematic mapping study on mining code comments. Recently, [22] performed a systematic mapping study to analyze studies on MSR by examining five editions of Working Conference on Mining Software Repositories (MSRConf). The work reported that comments analysis is one of the most explored objects of study.

Comments analysis can reveal information such as the reason for adding new lines of code, knowing the progress of a collective task, or even why relevant changes were performed. Besides, code comments may describe the developers' point of view about quality issues in the software development [122]. We recognize the importance of the research efforts invested in the area on the development of new techniques to support comments analysis or new tools to extract or process the collected comments. However, it would be beneficial for the area to have a broad view of the current research that has been performed, so new directions of research could be better defined.

Although some systematic literature reviews have been performed in the MSR area [46][22][42][64], none of them has focused on comments analysis as object of study. In this context, this work presents the results of a mapping study performed to investigate the following research question: "*How has the comments analysis been explored with the purpose of supporting software engineering activities?*". By answering this question, we intend to identify which purposes, focuses, techniques, tools, research types and evaluation methods have been considered on the research on comment analysis to support software engineering.

B.1 SYSTEMATIC MAPPING METHOD

This work follows a well-organized set of guidelines for carrying out systematic methods in the context of software engineering [104]. This process comprises the following activities: definition of research questions, search strategy to gather relevant studies, data sources, inclusion and exclusion criteria for primary studies, screening of papers, data extraction, and data analysis.

B.1.1 Definition of Research Questions

For this study, a primary Research Question (RQ) was defined: "*How has the comments analysis been explored with the purpose of supporting software engineering activities?*". The following complementary research questions were derived from this main one. By answering these questions, we will have a detailed characterization of the identified studies.

RQ1. *Which are the main purposes and focus of researches on analysis of comments?*
The objective of this question is to identify the goal of MSR approaches in the area of analysis of comments. To perform the classification of the extracted data, we used a taxonomy discussed by [22], that is publicly available [123]. By identifying the purpose, we classify the primaries intentions of the studies (e.g., identification, characterization, prediction). In complement, by identifying their focuses (e.g., technical debt and defect), we classify the main attribute of interest between the purpose and the object of analysis (comment analysis). Thus, for example, we could have: to identify (purpose) technical debt (focus) exploring comments analysis (object of analysis).

RQ2. *What are the techniques used by researchers to analyze comments?*
This question intends to identify what techniques of comments analysis have been used to extract, process, and analyze comments. We also intend to categorize the techniques as manual, semiautomatic and automatic.

RQ3. *What are the tools used to extract, process, or analyze comments?*
This question aims to know what tools have been used to extract, preprocess and analyze comments. The result of this RQ is a set of tools that may be used in comment mining process. This information may support researchers to discover new tools, their finality, features, and how they may be explored to mining comments. This set of tools may also support them to create their own tool to explore comments.

RQ4. *Which empirical evaluations have been performed in the area?*
This question aims to identify whether the proposed approaches have been evaluated through empirical methods, and if so, which method was used. To classify the types of studies, we considered the empirical evaluations types discussed by [22]. The taxonomy is available at [123].

RQ5. *What are the research types identified?*
This question intends to categorize the studies according to the research type facets defined by [124]:

- **Evaluation Research**: techniques are implemented in practice and an evaluation of the technique is conducted;
- **Experience Papers**: explain on what and how something has been done in practice. It has to be the personal experience of the author;
- **Opinion Papers**: these papers express the personal opinion of somebody, whether

a certain technique is good or bad or the way things should be done. They do not rely on related work and research methodologies;

- **Philosophical Papers**: these papers sketch a new way of looking at existing things by structuring the field in form of a taxonomy or conceptual framework;
- **Solution Proposal**: a solution for a problem is proposed, the solution can be either novel or a significant extension of an existing technique. The potential benefits and the applicability of the solution is shown by a small example or a good line of argumentation;
- **Validation Research**: techniques investigated are novel and have not yet been implemented in practice;

With the answer for this RQ, we identify the overall contribution provided by the studies and, in combination with the answers of RQ4, it allows the identification of gaps and needs in the area.

B.1.2 Search strategy

We first chose the following keywords to perform the search in the digital libraries:

- **Population**: software, system, program, application;
- **Intervention**: analysis, identification, detection, examination, study, code, comment;
- **Outcome**: methodology, technique, method, practice, process, strategy, tool.

However, after some tests performed in digital libraries, we observed that the search string was too restrictive, returning a small number of papers. This could result in an incomplete mapping process. Therefore, we chose a more general search strategy, using only the keywords of population and intervention:

((Software OR System OR Program OR Application) AND ((Comment analysis) OR (Comment Identification) OR (Code Comments) OR (Comment detection) OR (Examination of comments) OR (Analysis of comment) OR (Comments analysis) OR (Study of comments) OR (Comments study)))

We applied this search string to Titles and Abstracts. We chose not to do full text search because we found that full text search resulted in a very large number of studies out of scope. In addition, we considered papers published until October 2016.

B.1.3 Data Source

In choosing data sources, we aimed to include important journals and conferences regarding the research topic. For this, we considered the list recommended by Brereton et al. [102]: ACM Digital Library, IEEE Xplorer, Science Direct, Engineering Village, Springer Link, Scopus, and Citeseer.

B.1.4 Study Selection

After applying the search strings in the digital libraries, we filtered the relevant primary studies from the search results using the following selection criteria:

- **Inclusion criteria**: (i) Published works that describe how comments are used in software engineering activities; (ii) Once several papers reported the same study, only the most recent were included; (iii) The publication date of the article should be between 1990 and 2016; and (iv) Papers published in journals, conferences or peer reviewed journals.

- **Exclusion criteria**: (i) Studies outside the scope of this research; (ii) Papers that are only available in the form of workshop/conference reports, abstracts or PowerPoint presentations; (iii) Duplicated papers; and (iv) Book chapters and articles published without revisions (white papers).

B.1.5 Screening of Papers

The screening of papers process to identify the primary studies comprises the following steps: (i) apply the selection criteria by reading the paper title and abstract and select the relevant studies from the search results; and (ii) read introduction and conclusion in case the researcher needs further information to decide on the study selection. Two researchers performed these steps and other two experienced researchers reviewed the results.

B.1.6 Data Extraction

Before the data extraction execution, we performed a pilot extraction, aiming to align the researchers' understanding of the research questions. During the extraction process, researchers carefully read the primary studies in a peer-review process. Two researchers extracted data for the same study and a third researcher solved the possible disagreements. All relevant data of each study was registered in a spreadsheet. At the end, one experienced researcher reviewed the extracted data. The complete data are publicly available [123].

B.1.7 Data Analysis and Synthesis

We considered a quantitative method to analyze the extracted data. Although we have done an analysis on the results, most of them were summarized to present an overview of the findings. Thus, this work is characterized as a scoping study, which maps the primary studies on mining code comments in the software engineering area.

B.2 RESULTS AND DISCUSSION

We selected the papers following four steps. First, we searched papers in the digital libraries, resulting in 402 studies. Then, we removed the duplicated papers, resulting in

240. Next, we read the title and abstract to remove the studies out of this work scope. This activity resulted in 68 studies, which were fully read. During this last step, we removed 32 studies, resulting in 36 papers to perform the data extraction. The complete list of the studies is available at [123].

We have 24 papers published on conferences, 9 on journals, and 3 on workshops. From a temporal point of view (Figure B.1), we identified an increasing number of publications since 2011. We also observed that 2015 was the year with more publications (6 – 16.7%), followed by 2016 (5 – 13.9%), 2011 and 2014, both with the same number of studies (4 – 11.43%). The increasing number of publications in the last two years (2015 and 2016) is partially justified by the presence of works relating comments analysis and Quality of Software Artifacts/Technical Debt. We can observe this trend in Figure B.2.

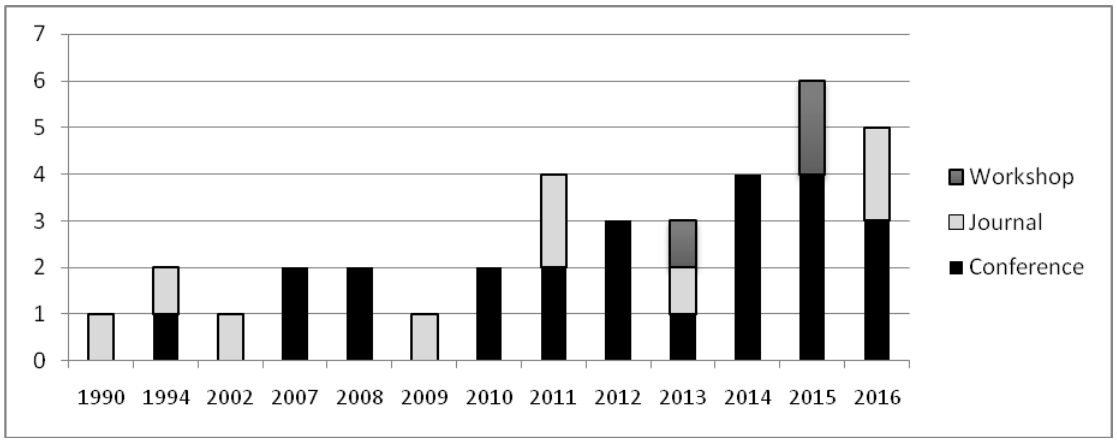


Figure B.1 Temporal View of Studies.

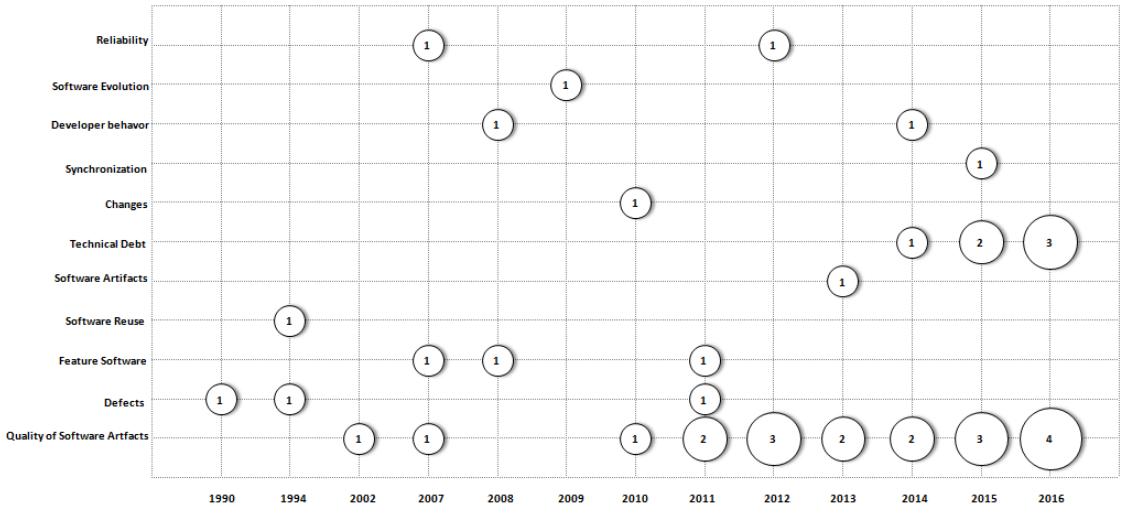


Figure B.2 Studies' focus over the years.

B.2.1 Purposes and focus of researches (RQ1)

To analyze RQ1 and classify the studies according their purpose and focus, we considered the taxonomy discussed in [22]. We also considered that each study could be categorized into more than one classification. For example, a study can explore comments to *identify* and *comprehend* (purpose) the quality of software artifacts (focus).

Purpose: "Comprehension" was considered in the majority of the studies (29 – 80.6%), followed by "identification" (10 – 27.8%). Studies related to the purpose comprehension intend to understand the behavior of a specific attribute by analyzing code comments (e.g., comprehension of the quality of software artifacts, using code comment analysis). Whereas, studies with the purpose identification aim to detect a specific attribute through comment analysis (e.g., to identify defect).

Only two studies had "improvement" as main purpose and all other categories ("classification", "evaluation", "localization", "association", and "characterization") were identified in only one study.

Focus: Figure B.2 shows the classification of the focus of the studies over the years. The majority of studies focused on "Quality of Software Artifacts" (19 – 52.8%) followed by "Technical Debt" (6 – 16.7%). "Quality of Software Artifacts" appeared nearly every year, but the number of studies with this focus has increased in the last years. The focus on "Technical Debt" only started to be considered more recently, in 2014, and it has also increased in the last years, revealing a new area of investigation.

Purpose x Focus: Figure B.3 presents a bubble chart showing the relationship between the facets purpose and focus. We can observe that "Comprehension" of "Quality of Software Artifacts" is the most explored purpose and focus in 16 out of 36 studies. The second most identified purpose and focus is "Identification" of "Quality of Software Artifacts", "Comprehension" of "Technical Debt", and "Identification" of "Technical Debt" with 4 studies each.

The results also pointed out that, while the purposes "Evaluation", "Localization", "Improvement", "Classification", "Association", and "Characterization" explore just one focus, the purpose "Comprehension" explores almost all focuses identified in this work. Regarding technical debt, we can observe that the studies on the area have been performed with the purposes of "Comprehension" or "Identification".

B.2.2 Techniques to analyze comments (RQ2)

To answer RQ2, we analyzed the techniques that have been used to mine comments. We identified 14 techniques at total and observed that 80.6% of the studies used the following three: Dictionary/Vocabulary (13 studies – 36.1%), NLP (10 – 27.8%), and Statistic/Statistic Analysis/Method Statistic (6 – 16.7%).

Figure B.4 shows the relationship between the study purpose and the technique used to explore comments analysis. "Comprehension and Dictionary/Vocabulary" were the purpose and technique most used together. Next, "NPL" was used together with "Comprehension and Identification". In the following, we have "Identification and Dictionary/Vocabulary". We can also observe that "Comprehension and Identification" were combined with almost all techniques. On the other hand, some techniques have been

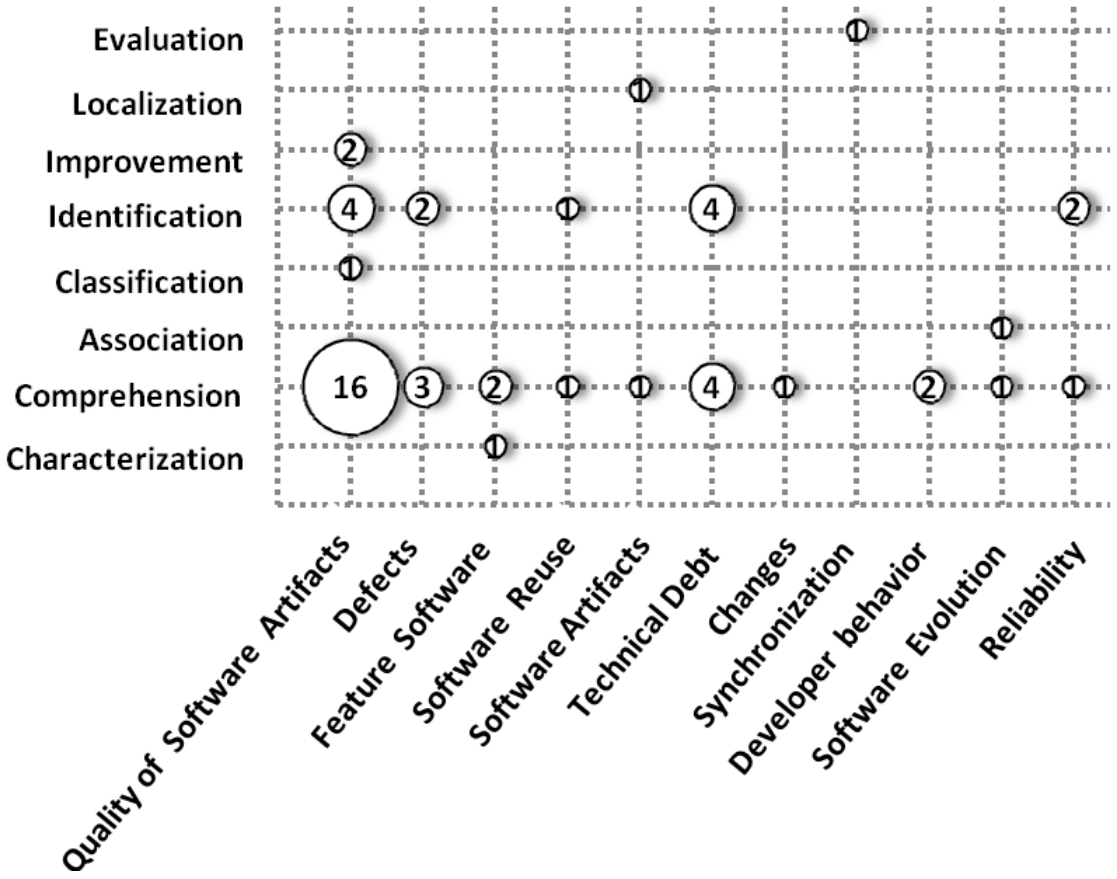


Figure B.3 Purpose x Focus.

used by only one study (e.g. Dynamic analysis and clustering).

By analyzing the way each technique works, we also identified that 55.6% of them were semiautomatic, followed by automatic (27.8%), manual (8.3%), and the other 8.3% was not determined. A possible reason for the low usage of manual techniques can be the cost to perform a manual analysis in terms of effort and also the fact that the process would be error prone.

B.2.3 Tools used to extract, process, or analyze comments (RQ3)

Table B.1 presents the tools identified in this work by each step of a mining process (extraction, processing, and analysis). We identified 16 tools used in the extraction step, 14 for the analysis, and 2 for the processing. We did not identify any tool in 50% of the selected studies.

By analyzing Table B.1, we can see that only one tool is used in more than one step (iComment). The others developed to support only one step of the comment mining process. We also identified that most of the tools are only presented in one study. Therefore,

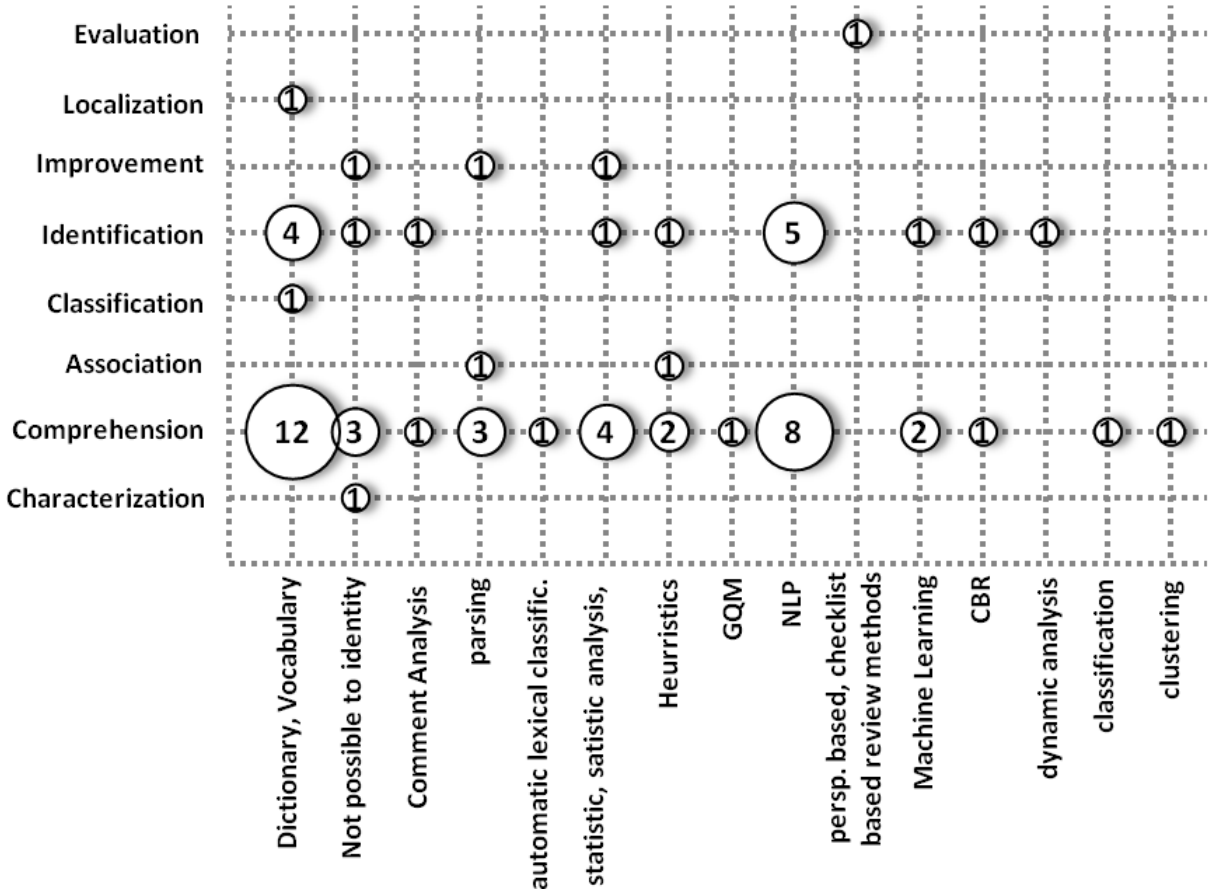


Figure B.4 Studies purpose per technique.

in general, researchers develop new tools as a result of their work. A possible explanation for this is that each study has a specific need (not considered in existing tools) of exploring comments in order to achieve its goals.

B.2.4 Empirical evaluations (RQ4)

In this mapping study, we found that 16 papers (44.4%) performed controlled experiments, 8 (22.2%) case studies, 4 (11,1%) exploratory studies, and only 1 study (2.8%) performed a survey. We also identified 1 ethnographic study (2.8%). Finally, 6 (16.7%) studies did not present any evaluation. This result indicates that the works in the area of software comments analysis are characterized by the use of empirical methods to assess the proposed approaches.

Figure B.5 represents the evaluation methods performed per purpose. We can observe that controlled experiment and case study are the main research methods used for evaluating "Comprehension" and "Identification" tasks. "Classification", "Evaluation" and "Characterization" were the three categories in which researchers have not used empirical methods to evaluate their approaches.

Table B.1 List of tools and mining step

Step	Tool
Extraction (16)	CLOC, tComments, @Randoop, Prototype Tool, RBG tool, SLOCCount, Jdeodorant, srcML, SSLdoclet, ConQAT, C-REX, Evolizer and ChangeDistiller, eXcomment, iComments, JavaMethodExtractor
Processing (2)	iComments, LI Tools
Analysis (14)	tComments,@Randoop, QDA Analysis tool, RBG tool, CommentCounter, LOCCounter, COMTOR, Evolizer and ChangeDistiller, iComments, Javadocminer, MineHEAD , Stanford Parser, next word prediction tool

B.2.5 Research types (RQ5)

Considering the taxonomy of research types presented in [124], we found that the majority of studies were a "Solution Proposal" paper (19 – 52.8%), 12 were a "Evaluation Research" (33.3%), and 3 were a combination of "Solution Proposal" with "Evaluation Research" (8.3%). Only 1 study performed a "Validation Research" (2.8%) and 1 study used "Opinion Research" (2.8%). Solution proposal is paper where a solution for a problem is proposed. The potential benefits and the applicability of the solution is shown by a small example or a good line of argumentation. On the other side, evaluation research is type of paper in which techniques are implemented in practice and an evaluation of the technique is conducted.

Figure B.6 presents the distribution of the performed research types over the years. We can observe that while the number of "Solution Proposals" published during the years is stable, there is a rising number of "Evaluation Research" in the last few years (2014, 2015 and 2016) indicating a tendency in the area to perform and report more empirical studies.

Figure B.7 presents the relationship between research type and focus. It shows that the types "Solution Proposal" and "Evaluation Research" were widely adopted by researchers to investigate "Quality of Software Artifacts". We can also see that the focus "Technical Debt" is the second most explored considering these same types. The other types of research appear only as isolated initiatives.

B.2.6 Implications for Researchers and Practitioners

This work has mapped studies on MSR area that explored code comments. The list of studies, extracted data, identified gaps and findings can guide researchers in further studies in MSR area focused on code comments. As a consequence, this information can reduce the efforts of the researchers to find and select those studies related to the purposes and focuses of their research. It is also relevant for newcomers who are interested in exploring the code mining area.

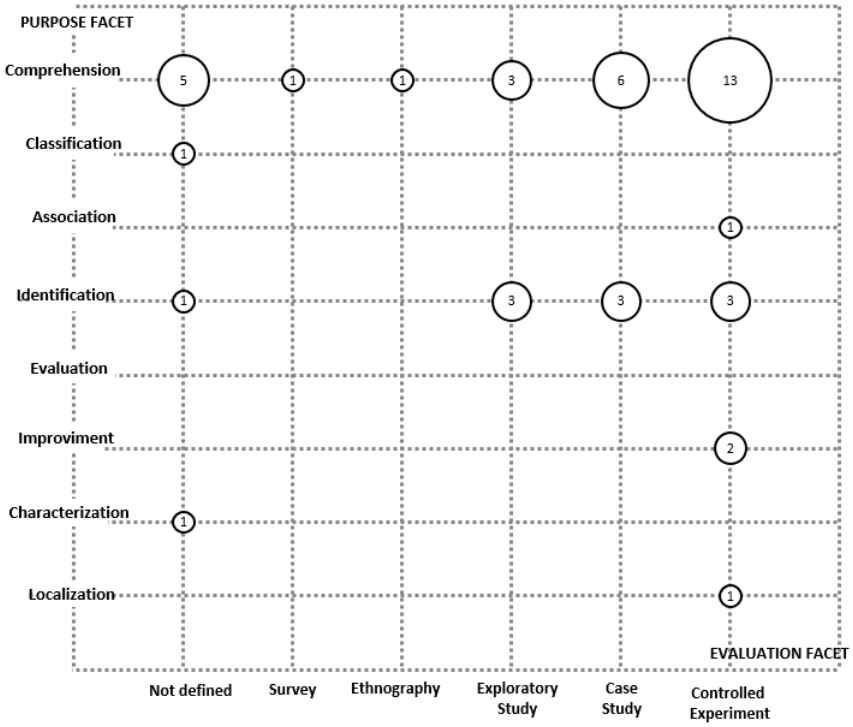


Figure B.5 Purpose vs Evaluation methods

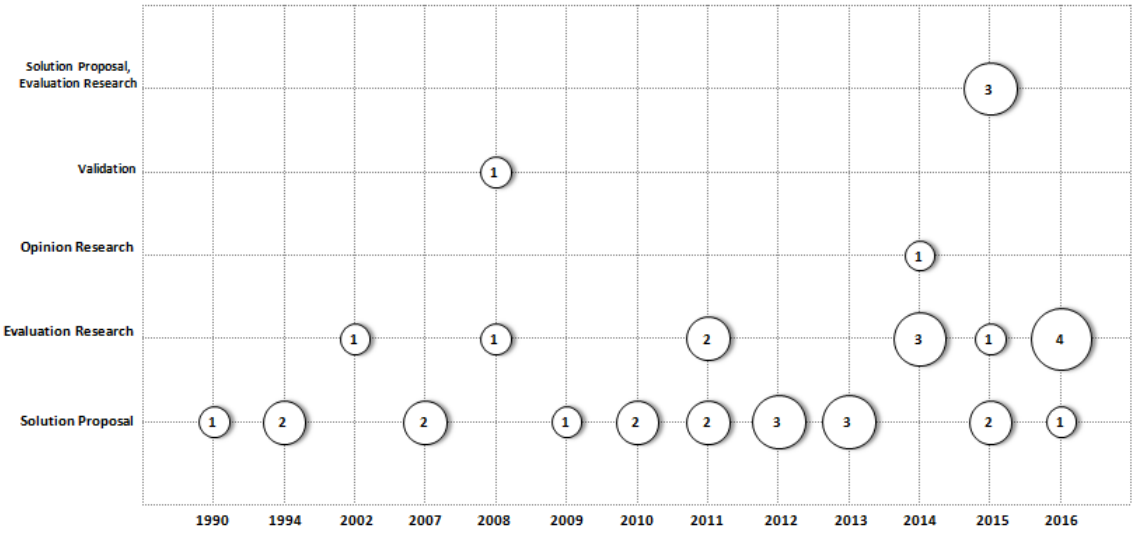


Figure B.6 Evolution of the research type over the years.

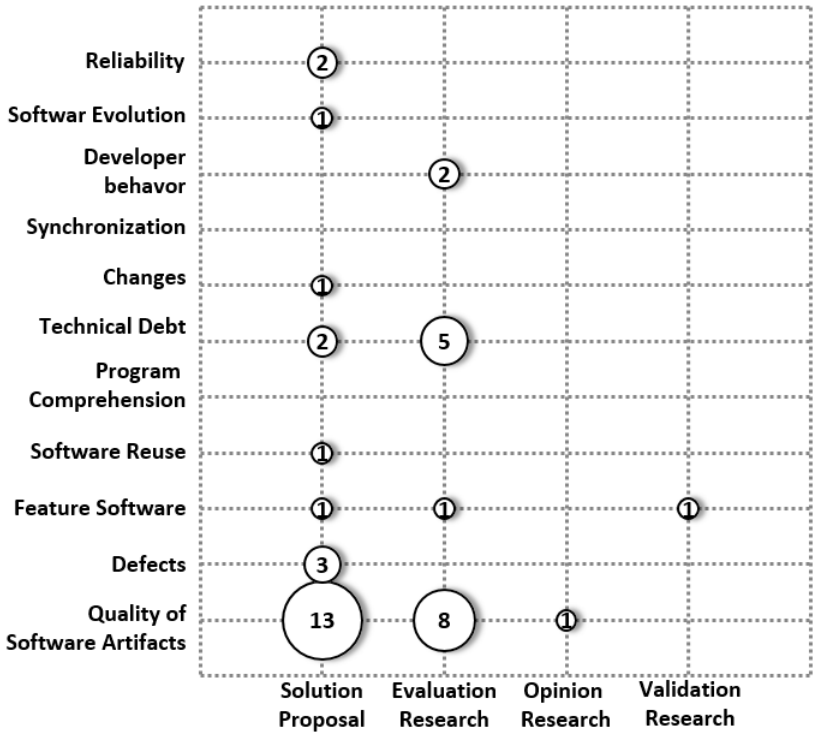


Figure B.7 Focus vs Research type.

We catalogued a set of tools and techniques used in the investigation studies to mine comments with several purposes. For practitioners, our study can help them to know these tools and techniques, avoiding developing approaches to extract, processing or analyzing comments with the same purposes. This knowledge can be used as a guide to implement different approaches to explore different purposes.

B.3 THREATS TO VALIDITY

This study has some threats to validity. We describe them and present strategies to mitigate their bias.

Selection Bias. We addressed this threat during the selection step of the studies. We selected each study based on the judgment of the inclusion and exclusion criteria. However, some studies could have been categorized incorrectly. With the intention of mitigating this threat, we discussed the study protocol among the researchers to guarantee a common understanding and we identified each study through a selection process which comprised multiple steps.

Construct validity. Construct validity measures the ability of the research to measure what was intended. The questions investigated in this study may not cover comment mining field in MSR area. We made an effort to address the main aspect of comment mining field. In order to try answering the research questions, we explored the same facets used by Farias et al [22] and defined by [23].

Data extraction. Data extraction bias may cause issues in the classifications of

facets and the analysis of selected studies. The data collection was performed manually, thus it can be an error prone activity. To reduce this bias, we used an electronic data extraction form in order to facilitate the data organization and subsequent analysis. The form was designed to collect all the information needed to address the review questions. The data extraction form was piloted on a sample of primary studies and doubts were discussed and solved in order to make clear the meaning of data items to all researchers. Next, two researchers conducted the data extraction process and mitigated conflicts in a consensus meeting. Finally, a third researcher analyzed the issues on each classification or extracted information to make sure that the extracted data were valid and clear for further analysis.

External validity. It considers the extent to which the effects observed in the study are generalized and applied outside of the study [104]. In this sense, we carried out a systematic mapping study over studies published until October 2016 that focused on comment mining. It implies that we might have missed some relevant studies. Thus, our findings may not be generalized for whole comment mining area, but our outcomes allow us to draw insights to guide further investigations that intend to analyze comments.

Appendix C

THE CONTEXTUALIZED VOCABULARY - FIFTH RELEASE

Pattern	Score	Level	Combination	Flag	TD types
TODO	2,00	not much decisive	Tag	1	0,00
FIXME	2,50	Decisive	Tag	1	Defect debt
XXX	2,00	not much decisive	Tag	1	0,00
HACK	2,00	not much decisive	Tag	1	0,00
REMIND	2,00	not much decisive	Tag	1	0,00
REVIEW	2,00	not much decisive	Tag	1	0,00
REVISIT	2,00	not much decisive	Tag	1	0,00
REMARK	1,50	not much decisive	Tag	1	0,00
CHECKME	2,00	not much decisive	Tag	1	0,00

Pattern	Score	Level	Combination	Flag	TD types
might be	2,00	not much decisive	OTV	1	0,00
have to	2,60	Decisive	OTV	1	0,00
must be	2,00	not much decisive	OTV	1	0,00
need to	2,50	Decisive	OTV	1	0,00
should be	2,50	Decisive	OTV	1	0,00
should not be	3,00	Decisive	OTV	1	0,00
should have	2,00	not much decisive	OTV	1	0,00
would be	2,00	not much decisive	OTV	1	0,00
would not be	3,00	Decisive	OTV	1	0,00
will not be	2,00	not much decisive	OTV	1	0,00
could be	1,50	not much decisive	OTV	1	0,00
would have	2,00	not much decisive	OTV	1	0,00
should never	2,00	not much decisive	OTV	1	0,00
must have	1,00	can be decisive	OTV	1	0,00
may have	3,00	Decisive	OTV	1	0,00
may return	3,00	Decisive	OTV	1	0,00
should allow	1,00	can be decisive	OTV	1	0,00
should use	2,00	not much decisive	OTV	1	0,00

Pattern	Score	Level	Combination	TD types
TODO	2,00	not much decisive	Tag	0,00
FIXME	2,50	Decisive	Tag	Defect debt
XXX	2,00	not much decisive	Tag	0,00
HACK	2,00	not much decisive	Tag	0,00
REMIND	2,00	not much decisive	Tag	0,00
REVIEW	2,00	not much decisive	Tag	0,00
REVISIT	2,00	not much decisive	Tag	0,00
REMARK	1,50	not much decisive	Tag	0,00
CHECKME	2,00	not much decisive	Tag	0,00

Pattern	Score	Level	Combination	TD types
might be	2,00	not much decisive	OTV	0,00
have to	2,60	Decisive	OTV	0,00
must be	2,00	not much decisive	OTV	0,00
need to	2,50	Decisive	OTV	0,00
should be	2,50	Decisive	OTV	0,00
should not be	3,00	Decisive	OTV	0,00
should have	2,00	not much decisive	OTV	0,00
would be	2,00	not much decisive	OTV	0,00
would not be	3,00	Decisive	OTV	0,00
will not be	2,00	not much decisive	OTV	0,00
could be	1,50	not much decisive	OTV	0,00
would have	2,00	not much decisive	OTV	0,00
should never	2,00	not much decisive	OTV	0,00
must have	1,00	can be decisive	OTV	0,00
may have	3,00	Decisive	OTV	0,00
may return	3,00	Decisive	OTV	0,00
should allow	1,00	can be decisive	OTV	0,00
should use	2,00	not much decisive	OTV	0,00

Pattern	Score	Level	Combination	TD types
can	1,50	not much decisive	Modal	0,00
shall	1,50	not much decisive	Modal	0,00
will	1,50	not much decisive	Modal	0,00
must	1,50	not much decisive	Modal	0,00
should	1,50	not much decisive	Modal	0,00
would	1,50	not much decisive	Modal	0,00
could	1,50	not much decisive	Modal	0,00
may	1,50	not much decisive	Modal	0,00
might	1,50	not much decisive	Modal	0,00

Pattern	Score	Level	Combination	TD types
Todo: ?	2,00	not much decisive	Tag + Question	0,00
Todo: Check	4,00	Very decisive	Tag + AV	0,00
TODO: Test	3,00	Decisive	Tag + AV	Test debt
TODO: Here too?	2,00	not much decisive	Tag + Adv	Design debt
TODO: Why	3,00	Decisive	Tag + question	0,00
Todo: What?	4,00	Very decisive	Tag + Question	Documentation debt
Todo: how to	1,50	not much decisive	Tag + Question	0,00
TODO: ?delete	2,00	not much decisive	Tag + Question + AV	0,00
TODO: Issue	2,00	not much decisive	Tag + Noun	0,00
Todo: Make	3,00	Decisive	Tag + expression	0,00
Todo: maybe	3,00	Decisive	Tag + Adv	0,00
Todo: The recurrence	3,00	Decisive	Tag + Noun	0,00
Todo: link to	3,00	Decisive	Tag + AV	0,00
TODO: Reuse	3,00	Decisive	Tag + AV	0,00

might be	2,00	not much decisive	OTV	0,00
Todo: temporary	4,00	Very decisive	Tag + Adj	Code debt, Design debt
Todo: Replace	3,50	Very decisive	Tag + AV	0,00
Todo: Should difine	3,50	Very decisive	Tag + Modal + Av	0,00
Todo: not implemented	4,00	Very decisive	Tag + Neg + Adj	Requirement debt
Todo: split	4,00	Very decisive	Tag + AV	0,00
Todo: move	3,00	Decisive	Tag + AV	0,00
Todo: verify	3,00	Decisive	Tag + AV	0,00
Todo: ??	1,00	can be decisive	Tag + Question	0,00
Todo: review	2,00	not much decisive	Tag + AV	0,00
Todo: Redo	4,00	Very decisive	Tag + AV	0,00
TODO: improve	3,00	Decisive	Tag + AV	Code debt
TODO not correct	3,00	Decisive	Tag + Adj	Requirement debt
Todo: Not	4,00	Very decisive	Tag + Neg	0,00
Todo: really should be	4,00	Very decisive	Tag + OTV	0,00
should not	4,00	Very decisive	modal + Neg	0,00
Todo: should have	3,50	Very decisive	Tag + OTV	0,00
should fail	3,00	Decisive	Modal + Verb	Defect debt
should probably go	3,00	Decisive	expression	0,00
Todo: Rework	3,00	Decisive	Tag + AV	Requirement debt
do all following steps	3,00	Decisive	expression	0,00
rather bad hash solution	3,50	Very decisive	Adj + Noun	Code debt, Design debt
bad idea	3,00	Decisive	Adj + Noun	Code debt
something bad	3,00	Decisive	expression	Code debt
Todo Use	2,00	not much decisive	Tag + Av	0,00
Todo: Can	2,00	not much decisive	Tag + modal Verb	0,00
need to change	2,00	not much decisive	OTV + AV	0,00
will be moving	2,00	not much decisive	OTV + AV	0,00
can be used instead	2,00	not much decisive	OTV + AV	0,00
slows things down	2,00	not much decisive	Expression	Code debt
could probably be	2,00	not much decisive	OTV + Adv	0,00
is needed	2,00	not much decisive	Verb	Requirement debt
Todo: how to handle	2,00	not much decisive	Tag + Av	0,00
can remove	2,00	not much decisive	Modal + AV	0,00
should attempt	2,00	not much decisive	Modal + Verb	0,00
public?	1,00	can be decisive	Question	0,00
cause exception	1,00	can be decisive	AV + Noun	Code debt
Note: code duplicates	4,00	Very decisive	Tag + Adj	Code debt, Design debt
Note: This class has gotten too big	3,50	Very decisive	expression	Design debt
Note: this try/catch block is copied fr	2,50	Decisive	expression	Code debt, Design debt
XXX: deprecated	3,50	Very decisive	Tag + Adj	Code debt
too large	2,00	not much decisive	Adj	Design debt
see bug	3,50	Very decisive	AV + Noun	Defect debt
seems to be a bug	3,50	Very decisive	expression	Defect debt
workaround the bug	4,00	Very decisive	expression	Defect debt
stupid error	3,00	Decisive	Adj + Noun	Defect debt
trial and error	2,50	Decisive	expression	Defect debt
fatal error	2,50	Decisive	Adj + Noun	Defect debt
TODO: would be better to refactor	3,00	Decisive	Tag + expression	Code debt
intentionally fails	3,00	Decisive	expression	Defect debt
test will fail	3,50	Very decisive	modal + AV	Test debt
Note: senseless	3,00	Decisive	Tag + Adj	Requirement debt
do not support	2,00	not much decisive	expression	Code debt
broken archive	2,00	not much decisive	Av + Noun	Code debt
This is broken	2,00	not much decisive	LV + AV	Code debt
incorrectly caches	2,50	Decisive	expression	Code debt
Note: incorrect here	4,00	Very decisive	Tag + Adj	Design debt
are not currently	1,50	not much decisive	LV + Adv	Code debt

might be	2,00	not much decisive	OTV	0,00
can never be	3,00	Decisive	expression	Code debt
Note: this is inefficient	4,00	Very decisive	Tag + Adj	Code debt
XXX: does this really belong here?	3,50	Very decisive	Tag + expression	Design debt
break the action	2,50	Decisive	Av + Noun	Code debt
dirty hack	3,50	Very decisive	Adj + Noun	Code debt
need this bullshit	3,50	Very decisive	expression	Code debt
bad hack	3,00	Decisive	Adj + Noun	Code debt
stupid hack	3,00	Decisive	Adj + Noun	Code debt
seems silly	2,50	Decisive	AV + Adj	Code debt
find a better place	3,00	Decisive	expression	Design debt
TODO: this has to be here	2,50	Decisive	Tag + expression	Design debt
could be more efficient	3,50	Very decisive	expression	Code debt
won't work	3,00	Decisive	modal + AV	Code debt
TODO broken	2,50	Decisive	Tag + AV	Code debt
should be enough	3,00	Decisive	OTV + Adv	Code debt
TODO labels?	2,50	Decisive	Tag + Question	0,00
TODO specification?	3,00	Decisive	Tag + Question	Documentation debt
Is this correct?	2,00	not much decisive	Question	0,00
Does this help?	1,00	can be decisive	Question	0,00
TODO: valid?	3,00	Decisive	Tag + Question	0,00
Todo: constraints	2,50	Decisive	Tag + Noun	0,00
must assemble	3,00	Decisive	Modal + AV	0,00
REVIEW Maybe this should be	3,50	Very decisive	Tag + OTV	0,00
need to re-evaluate	3,00	Decisive	OTV + AV	0,00
need to eliminate	3,00	Decisive	OTV + AV	0,00
Note: we should implement	4,00	Very decisive	Tag + Modal + Av	Requirement debt
could override this method	3,50	Very decisive	expression	Code Debt
should be overriden	3,00	Decisive	OTV + AV	0,00
Note: need	3,00	Decisive	Tag + OTV	0,00
Note: could eliminate	3,00	Decisive	Tag + Modal + Av	0,00
Note: should	2,50	Decisive	Tag + Modal	0,00
need to be rewritten	4,00	Very decisive	OTV + AV	Code debt
TODO: need to add	3,50	Very decisive	Tag + OTV + AV	0,00
XXX clean up	2,50	Decisive	Tag + AV	0,00
XXXX re-evaluate	3,00	Decisive	Tag + AV	0,00
Replace this later	3,00	Decisive	AV + adv	Code debt
hack to get fix	3,00	Decisive	expression	Defect debt
hack to move	3,00	Decisive	expression	0,00
hack to work	3,00	Decisive	expression	0,00
hack to avoid	3,00	Decisive	expression	0,00
must be a better way	3,00	Decisive	expression	Code debt
FIXME: check	3,00	Decisive	Tag + AV	Defect debt
FIXME: rename	3,00	Decisive	Tag + AV	Defect debt
FIXME: use	3,00	Decisive	Tag + AV	Defect debt
hack: find	3,00	Decisive	Tag + AV	0,00
TODO: refactor	3,50	Very decisive	Tag + AV	0,00
should make	2,50	Decisive	Modal + AV	0,00
need to validate	3,00	Decisive	OTV + AV	0,00
TODO: weird	3,00	Decisive	Tag + Adj	0,00
have to be careful	3,00	Decisive	OTV + Adj	Requirement debt
resolve this discrepancy?	2,50	Decisive	expression	0,00
Note: this implementation is tempora	4,00	Very decisive	Tag + expression	Code debt, Design debt
may be better	3,00	Decisive	expression	Code debt

might be	2,00	not much decisive	OTV	0,00
TODO: should add test	3,00	Decisive	Modal + Av	Test debt
Temporary hack	4,00	Very decisive	expression	Code debt, Design debt
TODO: maybe we should test	4,00	Very decisive	Tag + Modal + Av	Test debt
could decouple	2,50	Decisive	Modal + AV	Design debt
could fix	2,50	Decisive	Modal + AV	Defect debt
completely unnecessary	2,50	Decisive	expression	Code debt
will be longer	2,00	not much decisive	OTV + Adj	0,00
TODO: wtf	3,00	Decisive	Tag + Acronym	Code debt
cannot be called	2,50	Decisive	expression	Code debt
problem here	3,00	Decisive	Noun + Adv	0,00
Bad code	3,00	Decisive	Adj + Noun	Code debt
Is there no better way?	2,00	not much decisive	Expression	Code debt
TODO: Is this needed?	3,00	Decisive	Tag + Question	Code debt
Todo: Does this help?	1,50	not much decisive	Tag + Question	0,00
Would it not better be	2,00	not much decisive	Expression	Code debt
Find a better way to do this	3,00	Decisive	AV + Adj + Noun	Code debt
not possible to do better	3,00	Decisive	Neg + Adj + OTV	Code debt
Todo: Is this a good way?	3,00	Decisive	Tag + expression/Question	Code debt, Design debt
probably better ways to implement th	3,00	Decisive	Adv + Adj + Noun + Av	Code debt, Design debt
is not the correct way	3,00	Decisive	LV + Neg + Adj + Noun	Requirement debt
not entirely correct	3,00	Decisive	Neg + Adv + Adj	Requirement debt
other way to implement	3,00	Decisive	Modal + AV + Noun + AV	Code debt
could probably do better	3,00	Decisive	expression	Code debt
big arrays	2,00	not much decisive	Adj + Noun	Code debt
Clean Up	1,00	can be decisive	expression	0,00
clone code	2,50	Decisive	Adj + Noun	Code debt, Design debt
Future enhancement	3,00	Decisive	Adj + Noun	Code debt
Ignore the error	4,00	Very decisive	AV + Noun	Defect debt
It 's an error	4,00	Very decisive	LV + Noun	Defect debt
treat this as a soft error	3,00	Decisive	expression	Defect debt
strong coupling	3,90	Very decisive	Adj + Noun	Design debt
god class	3,00	Decisive	Adj + Noun	Design debt
Harder to fix	3,00	Decisive	expression	Defect debt
large class	3,00	Decisive	Adj + Noun	Design Debt
Never implemented	3,00	Decisive	expression	Requirement debt
to be implemented	3,00	Decisive	expression	Requirement debt
no sense	2,50	Decisive	expression	Requirement debt
not implemented	3,00	Decisive	expression	Requirement debt
not completely implemented	3,00	Decisive	expression	Requirement debt
not necessary	3,00	Decisive	expression	Code debt
not really needed	2,50	Decisive	expression	Code debt
Quality problem	3,00	Decisive	Adj + Noun	Code debt
Slow performance	3,00	Decisive	Adj + Noun	Architecture debt
test score for nothing	1,00	can be decisive	expression	Test debt
tired to think	2,00	not much decisive	Adj + AV	People Debt
Unnecessary code	2,00	not much decisive	Adj + Noun	Code debt
Violation of modularity	4,00	Very decisive	expression	Architecture debt
What?	3,00	Decisive	Question	0,00
What do we do	2,00	not much decisive	Question	0,00
What about?	2,00	not much decisive	Question	0,00
Don't appear to be used	3,00	Decisive	Expression	Code debt
not used in the new APIs	2,00	not much decisive	Neg + AV + Noun	Code debt

might be	2,00	not much decisive	OTV	0,00
not used	2,00	not much decisive	expression	Code debt
don't use	2,00	not much decisive	expression	0,00
code is not used	1,00	can be decisive	Expression	Code debt
no faults are fixed	3,00	Decisive	expression	Defect debt
not work	3,50	Very decisive	expression	Code debt
Todo: This does not work	4,00	Very decisive	Tag + expression	Code debt
TODO: check if this works	4,00	Very decisive	Tag + AV + AV	Code debt
cyclic dependency	3,00	Decisive	expression	Architecture debt, Build debt
dependency cycle	2,00	not much decisive	Expression	Architecture debt, Build debt
Why is this not done	2,00	not much decisive	Question	Requirement debt
Todo: Why do I need to do this?	3,00	Decisive	Tag + Question	0,00
Why are these gone?	3,00	Decisive	Question	0,00
Todo: Why is this here?	4,00	Very decisive	Tag + Question	Design debt
why synchronized?	2,00	not much decisive	Question	0,00
Todo: Why was	2,50	Decisive	Tag + Question	0,00
TODO: why is this inside a block?	3,00	Decisive	Tag + Question	Design debt
Todo: Why does it fail	4,00	Very decisive	Tag + expression	Defect debt
causes and exception	4,00	Very decisive	expression	Code debt
not seem to define this name	1,00	can be decisive	expression	0,00
show some documentation?	3,00	Decisive	Av + Noun	Documentation debt
Not are useful here	2,00	not much decisive	Neg + LV + Adj	Design debt
nobody is using	3,00	Decisive	Expression	Code debt
nobody ever looks	3,00	Decisive	Expression	Code debt
is there a problem	2,50	Decisive	expression	0,00
I don't think	2,00	not much decisive	Expression	0,00
For debugging	2,00	not much decisive	expression	0,00
let's be sure	1,50	not much decisive	expression	Requirement debt
I don't understant	2,50	Decisive	expression	Documentation debt
not sure why	3,00	Decisive	Question	0,00
It causes the feedback loop	1,00	can be decisive	AV + Noun	Code debt
must be re-introduced to solve	2,00	not much decisive	OTV + AV	0,00
may be optimised	3,00	Decisive	OTV + AV	Code debt
Is there any other way?	2,00	not much decisive	expression	Code debt
not the right location	2,50	Decisive	expression	Design debt
Will be able	2,00	not much decisive	Expression	0,00
probably redundant	3,00	Decisive	Adv + Adj	Code debt, Design debt
it is used in several places	2,00	not much decisive	LV + AV + Noun	Code debt, Design debt
clone of this code	2,00	not much decisive	Expression	Code debt, Design debt
Need to Copy	2,00	not much decisive	OTV + AV	Code debt, Design debt
chose not to do this	2,00	not much decisive	expression	0,00
Todo: Which is best?	3,00	Decisive	Tag + question	Code debt
This is a bug	4,00	Very decisive	LV + Noun	Defect debt
cause issue	4,00	Very decisive	AV + Noun	Code debt
not according	2,00	not much decisive	expression	Requirement debt
is not a valid one	4,00	Very decisive	LV + Neg + Verb	debt
is slightly different than what's docur	3,00	Decisive	Expression	Documentation debt
cause problem	2,50	Decisive	AV + Noun	Code debt
still has some problems	4,00	Very decisive	Expression	0,00
Todo: does not handle	3,00	Decisive	Tag + Expression	0,00
Todo: Handle	2,00	not much decisive	Tag + AV	0,00
TODO: I don't think it's normal	2,50	Decisive	Tag + Expression	Requirement debt
Todo: has a bad smell	4,00	Very decisive	Tag + expression	Design debt
THIS IS A HACK	4,00	Very decisive	expression	Code debt

might be	2,00	not much decisive	OTV	0,00
memory cosuming	4,00	Very decisive	Expression	Build debt
Todo: This needs more work!	4,00	Very decisive	Tag + OTV + noun	Code debt
Todo: does not work	4,00	Very decisive	Tag + expression	Code debt
to be safe	3,00	Decisive	LV + Verb	Requirement debt
take care	3,00	Decisive	expression	Requirement debt
inherited problems	3,00	Decisive	AV + Noun	0,00
problems directly in	3,00	Decisive	expression	0,00
contains some errors	4,00	Very decisive	expression	Defect debt
FIXME: this could be a problem	3,00	Decisive	Tag + OTV + Noun	Defect debt
Low code coverage	3,00	Decisive	Adj + Noun	Test debt
Bad coding practies	4,00	Very decisive	Adj + Noun	Code debt
Time consuming	2,50	Decisive	expression	Build debt
Deficiency in testing activities	4,00	Very decisive	expression	Test debt
necessity of understanding	2,00	not much decisive	expression	Documentation debt
Work in progress here	4,00	Very decisive	expression	0,00
Note: This is temporary	4,00	Very decisive	Tag + LV + Adv	Code debt, Design debt
TODO: maybe should stop	4,00	Very decisive	Tag + Adv + Modal + Av.	0,00
need to be in their own files?	4,00	Very decisive	OTV + Noun	Design debt
A much better way	2,00	not much decisive	Adv + Adj + Noun	Code debt
defined in the ToDoItem	4,00	Very decisive	Expression	0,00
changes the complete structure	3,00	Decisive	AV + Noun	Design debt
Who is calling this?	3,00	Decisive	Question	Code debt
manually created todo item	2,00	not much decisive	Expression	0,00
Todo: This probably belongs	3,00	Decisive	Tag + Adv	Architecture debt
needs documenting	4,00	Very decisive	expression	Documentation debt
Information that is required	3,00	Decisive	Noun + To Be + Adj	Documentation debt
replace the deprecated	4,00	Very decisive	AV + Adj	Code debt
currently not used	1,50	not much decisive	expression	Code debt
not support Java 1.3 any more	4,00	Very decisive	Neg + AV+ Noun + adv	Code debt
need to be fixed	4,00	Very decisive	OTV + AV	Defect debt
Todo: We could also support	4,00	Very decisive	Tag + Modal + Av	0,00
problems directly in this class	3,00	Decisive	Noun + Adv + Noun	Code debt
need to do anything here?	2,00	not much decisive	OTV + Adv	0,00
good solution?	2,00	not much decisive	Question	0,00
Todo: improve the fig finding algorith	4,00	Very decisive	Tag + Av + Noun	Code debt
What to do	3,00	Decisive	Question + OTV	0,00
Note: maybe	2,50	Decisive	Tag + Adv	0,00
Todo: correctly implement	4,00	Very decisive	Tag + Adv + AV	Requirement debt
Todo: This is a temporary method	4,00	Very decisive	Tag + Adj + Noun	Code debt, Design debt
I don't understand	2,00	not much decisive	expression	Documentation debt
Todo: This is not according	4,00	Very decisive	Tag + LV + Adv	Requirement debt
Todo: This needs work	4,00	Very decisive	Tag + OTV	Code debt
unwanted dependency	3,00	Decisive	Expression	Architecture debt, Build debt
all memory allocated	4,00	Very decisive	Noun + AV	Build debt
to free memory	2,50	Decisive	expression	Build debt
bug workaround	4,00	Very decisive	expression	Defect debt
Workaround for a bug	4,00	Very decisive	expression	Defect debt
this sucks	4,00	Very decisive	Expression	Code debt
timing-dependent	2,00	not much decisive	expression	Architecture debt, Build debt

might be	2,00	not much decisive	OTV	0,00
TODO: maybe we should resolve	3,00	Decisive	Tag + Modal + Av	0,00
TODO: maybe take here?	2,50	Decisive	Tag + Question	Design debt
TODO maybe it would make	2,00	not much decisive	Tag + Modal + Av	0,00
should never happen	3,50	Very decisive	Modal + Adv + AV	Requirement debt
will never be	3,00	Decisive	Modal + AV	Requirement debt
makes no sense	3,00	Decisive	expression	Requirement debt
known problem	3,00	Decisive	Adj + Noun	0,00
is not enough	2,00	not much decisive	expression	Code debt
make sense?	2,50	Decisive	Question	0,00
broken down	2,00	not much decisive	expression	Code debt
horribly Inefficient	4,00	Very decisive	Adv + Adj	Code debt
consume very large	3,00	Decisive	Av + Adj	Architecture debt
modular violation	3,00	Decisive	expression	Architecture debt
large coupled	3,00	Decisive	Expression	Design debt
Not recommended	1,00	can be decisive	Expression	0,00
not sure	2,00	not much decisive	Expression	0,00
Problem of performance	4,00	Very decisive	Expression	Architecture debt
needes to be cleared	4,00	Very decisive	OTV + Adj	0,00
need to factor	3,00	Decisive	OTV + AV	0,00
Should be overriden	3,00	Decisive	OTV + AV	Code debt
no longer be used	3,00	Decisive	Expression	Code debt
barely understood	3,00	Decisive	expression	Documentation debt
bad smell	3,50	Very decisive	Adj + Noun	Design debt
deprecated code	3,50	Very decisive	Adj + Noun	Code debt
deprecated api	3,50	Very decisive	Adj + Noun	Code debt
would be better	3,00	Decisive	OTV + Adj	Code debt
probably a bug	4,00	Very decisive	expression	Defect debt
Todo: Add implementation	4,00	Very decisive	Tag + AV + Noun	Requirement debt
Todo: Add	3,00	Decisive	Tag + AV	Requirement debt
Todo: Fix	3,50	Very decisive	Tag + AV	Defect debt
Remove All	1,00	can be decisive	expression	0,00
remove dependencies later	2,00	not much decisive	Av + Noun + Adv	Architecture debt, Build debt
Remove the dependent	3,00	Decisive	AV + Noun	Architecture debt, Build debt
remove literal	2,00	not much decisive	AV + Noun	0,00
move to the correct owner	3,00	Decisive	expression	Design debt
Todo: Change	3,00	Decisive	Tag + AV	0,00
Need to be amended	3,00	Decisive	OTV + AV	0,00
Need to be updated	3,50	Very decisive	OTV + AV	0,00
Not created yet	2,00	not much decisive	Neg + AV	Requirement debt
deprecated call	3,50	Very decisive	Adj + Noun	Code debt
to be verified	2,00	not much decisive	LV + AV	0,00
Todo: Check the name	4,00	Very decisive	Tag + AV + Noun	0,00
maybe can be implemented	3,00	Decisive	Adv + modal + AV	0,00
is not yet completely implemented	4,00	Very decisive	Expression	Requirement debt
not yet used	1,00	can be decisive	expression	Code debt
is not updated	3,00	Decisive	Neg + AV	0,00
how do we resolve this?	3,00	Decisive	Question	0,00
TODO: resolve item	4,00	Very decisive	Tag + AV + Noun	0,00
has to be	2,00	not much decisive	OTV	0,00
have to set	3,00	Decisive	OTV + AV	0,00
have to move	3,00	Decisive	OTV + AV	0,00
Have to add	3,00	Decisive	OTV + AV	0,00
TODO: Do we really have to test	3,00	Decisive	Tag + OTV + SE Noun	Test debt
have to delete it later	4,00	Very decisive	OTV + AV + Adv	Code debt

might be	2,00	not much decisive	OTV	0,00
Have to do	3,00	Decisive	OTV + AV	0,00
must be resized	4,00	Very decisive	OTV + AV	0,00
need to clone	3,00	Decisive	OTV + AV	0,00
need to refactor	4,00	Very decisive	OTV + AV	Code debt
need to set	2,65	Decisive	OTV + AV	0,00
Need to be reviewed	4,00	Very decisive	OTV + AV	0,00
Need to be updated	3,50	Very decisive	OTV + AV	0,00
worn place	4,00	Very decisive	Adj + Noun	Design debt
need to be moved someplace useful	4,00	Very decisive	Expression	Design debt
to be moved	3,00	Decisive	LV + AV	0,00
need to be	3,50	Very decisive	OTV	0,00
Todo: Not sure we need to do this	4,00	Very decisive	Tag + OTV	0,00
need to jump	1,00	can be decisive	OTV + AV	0,00
should be visible	1,00	can be decisive	OTV + Verb	0,00
Why should it pass?	3,00	Decisive	Question	0,00
should be taken out	3,00	Decisive	Expression	Code debt
I don't think	2,00	not much decisive	Expression	0,00
should be fixed	4,00	Very decisive	OTV + AV	Defect debt
This should NOT be looking	2,00	not much decisive	Neg + OTV	0,00
should actually not be placed here	4,00	Very decisive	OTV + Adv	Design debt
should not be necessary	2,00	not much decisive	OTV + Adj	Code debt
should not have	3,00	Decisive	OTV	0,00
not to do this to keep architecture sta	2,00	not much decisive	Expression	Design debt
doesn't do anything	2,00	not much decisive	expression	Code debt
Will be needed	3,00	Decisive	Modal + To be	0,00
shall never	4,00	Very decisive	modal + Adv	0,00
Should find	2,00	not much decisive	Modal + AV	0,00
should handle	3,00	Decisive	Modal + AV	0,00
Should modify	4,00	Very decisive	Modal + AV	0,00
here?	3,00	Decisive	Question	Design debt
Todo: Here we should	3,00	Decisive	Tag + adv + Modal	0,00
Will be hard work	3,00	Decisive	OTV + Adj + Noun	Code debt
arbitrary dimensions	1,00	can be decisive	Adj + Noun	Code debt
somewhat inconsistent	4,00	Very decisive	Expression	Code debt
method is unsafe	3,00	Decisive	Noun + Adj	Code debt, Requirement debt
Dirty fix	4,00	Very decisive	Adj + Noun	Code debt
temporary solution	4,00	Very decisive	Adj + Noun	Code debt, Design debt
temporary crutch	4,00	Very decisive	Adj + Noun	Code debt, Design debt
possible future improvement	3,00	Decisive	Adj + Noun	Code debt
future maintenance	3,00	Decisive	Adj + Noun	0,00
future potential optimizations	3,00	Decisive	Adj + Noun	Code debt
Future release	3,00	Decisive	Adj + Noun	0,00
in the Future	1,00	can be decisive	Adv	0,00
unnecessarily complicated	3,00	Decisive	Adv + Adj	Code debt
This case is complicated	4,00	Very decisive	Noun + LV + Adj	Code debt
Todo: required	3,00	Decisive	Tag + Adj	Requirement debt
TODO: Is this required?	2,00	not much decisive	Tag + Question	0,00
Remove duplicates	2,00	not much decisive	AV + Noun	Code debt, Design debt
not quite right	2,00	not much decisive	Expression	0,00
isn't very solid	2,00	not much decisive	Expression	Code debt
get confused	2,00	not much decisive	Expression	Code debt
confusing name	2,50	Decisive	Adj + Noun	Code debt
Code so long?	3,00	Decisive	Question	Design debt
difficult to maintain	4,00	Very decisive	Expression	Code debt
extra work	4,00	Very decisive	Adj + Noun	Code debt
perhaps move to	2,00	not much decisive	Adv + AV	0,00
fails the test	3,00	Decisive	Verb + Noun	Test debt
can fail	2,50	Decisive	Modal + AV	Code debt

might be	2,00	not much decisive	OTV	0,00
assert fails	3,00	Decisive	Expression	Defect debt
complete fail	2,50	Decisive	AV + Noun	Defect debt
never called	1,00	can be decisive	Adv + AV	Code debt
never have	2,00	not much decisive	expression	0,00
repeated code	2,50	Decisive	Adj + Noun	Code debt, Design debt
rather bad	3,00	Decisive	Adv + Adj	0,00
alternative implementation	2,00	not much decisive	Adj + Noun	Code debt, Design debt
very expensive	2,50	Decisive	Adv + Adj	Code debt
Note: loose type	2,00	not much decisive	Tag + Adj + Noun	0,00
Note: kind of far-fetched	2,00	not much decisive	Tag + expression	0,00
Note: This class is messy	3,00	Decisive	Tag + Noun + LV + Adj	Design debt
XXX: hairy code	4,00	Very decisive	Tag + Adj + Noun	Code debt
Should be careful	3,00	Decisive	OTV + AV	Requirement debt
plainly impossible	3,50	Very decisive	Expression	Code debt
binding bug	3,00	Decisive	Expression	Defect debt
This is not a sufficient	3,00	Decisive	Expression	Requirement debt
weird hack	2,50	Decisive	Expression	Code debt
can be expensive	2,50	Decisive	OTV + Adj	Code debt
This could possibly hurt performance	3,00	Decisive	Expression	Architecture debt
is not reliable as a unit test	3,00	Decisive	Expression	Test debt
fully opaque	3,00	Decisive	Adv + Adj	Code debt
seems wrong	3,00	Decisive	AV + Adj	Code debt
something's gone wrong	3,00	Decisive	Expression	Code debt
Todo: implement this	4,00	Very decisive	Tag + AV	Requirement debt
Todo: Party overlaps	4,00	Very decisive	Tag + expression	0,00
Todo: Update	4,00	Very decisive	Tag + AV	0,00
can be deleted	3,00	Decisive	OTV + AV	0,00
clone method	1,00	can be decisive	Verb + Noun	Code debt, Design debt
have to use	3,00	Decisive	OTV + AV	0,00
later use	2,00	not much decisive	Adv + AV	0,00
must be moved	3,00	Decisive	OTV + AV	0,00
must come before	3,00	Decisive	OTV + Adv	Design debt
must notify modification	2,00	not much decisive	Modal + AV	0,00
need to check	1,00	can be decisive	OTV + AV	0,00
need to extend	3,00	Decisive	OTV + verb	0,00
need to listen	2,00	not much decisive	OTV + AV	0,00
need to verify	2,00	not much decisive	OTV + AV	0,00
TODO: indicate the direction!	2,00	not much decisive	Tag + AV + Noun	0,00
will be useful	2,00	not much decisive	OTV + Adj	0,00
would seem to imply	2,00	not much decisive	expression	0,00
will be replaced	2,00	not much decisive	OTV + AV	Code debt
update the model	1,00	can be decisive	Av + Noun	0,00
Calls are ORDER DEPENDENT	3,00	Decisive	Noun + Adj	Architecture debt, Build debt
Do we need	3,00	Decisive	Question	0,00
loops are double	3,00	Decisive	Noun + LV + Adj	Code debt
may be no need	1,00	can be decisive	expression	Code debt
now remove the pool, too	4,00	Very decisive	Expression	0,00
reduce the number	2,00	not much decisive	AV + Noun	0,00
references to elements of other pack	1,00	can be decisive	Question	Design debt
Should define	2,50	Decisive	Modal + AV	0,00
will cause	3,00	Decisive	Modal + AV	0,00
good idea?	3,00	Decisive	Question	0,00
better idea?	3,00	Decisive	Question	0,00
need to do	2,50	Decisive	OTV + AV	0,00

might be	2,00	not much decisive	OTV	0,00
at a loss	2,00	not much decisive	Expression	0,00
remove this code	3,00	Decisive	Expression	Code debt
hang our heads in shame	3,00	Decisive	Expression	0,00
get rid	2,50	Decisive	Expression	Code debt
doubt that this would work	3,00	Decisive	Expression	Code debt
this is bs	3,00	Decisive	Expression	Code debt
just abandon it	2,50	Decisive	expression	Code debt
hope everything will work	2,50	Decisive	expression	Code debt
certainly buggy	3,50	Very decisive	expression	Defect debt

Pattern	Score	Level	Combination	TD types
merged	1,50	not much decisive	AV	0
add	1,50	not much decisive	AV	0,00
fix	2,00	not much decisive	AV	Defect debt
remove	1,50	not much decisive	AV	0,00
move	3,00	Decisive	AV	0,00
change	1,00	can be decisive	AV	0,00
create	1,50	not much decisive	AV	0,00
generate	1,50	not much decisive	AV	0,00
store	1,50	not much decisive	AV	0,00
configure	1,50	not much decisive	AV	0,00
work	1,50	not much decisive	AV	0,00
iterate	1,50	not much decisive	AV	0,00
read	1,50	not much decisive	AV	0,00
modify	1,50	not much decisive	AV	0,00
Replace	2,00	not much decisive	AV	0,00
Verify	1,50	not much decisive	AV	0,00
improve	1,00	can be decisive	AV	Code debt
implement	1,50	not much decisive	AV	0,00
Support	1,50	not much decisive	AV	0,00
Repair	3,00	Decisive	AV	0,00
Rework	2,00	not much decisive	AV	Code debt
Cause	1,50	not much decisive	AV	0,00
update	1,50	not much decisive	AV	0,00
Try	1,50	not much decisive	AV	0,00
Solve	1,50	not much decisive	AV	0,00
Optimise	1,50	not much decisive	AV	Code debt
Copy	1,50	not much decisive	AV	0,00
rebuild	2,00	not much decisive	AV	Code debt
Resolve	1,50	not much decisive	AV	0,00
Indicate	1,50	not much decisive	AV	0,00
Reuse	1,00	can be decisive	AV	0,00
Split	1,50	not much decisive	AV	0,00
Redo	1,50	not much decisive	AV	0,00
jump	1,50	not much decisive	AV	0,00
rename	2,00	not much decisive	AV	0,00
allocate	1,50	not much decisive	AV	0,00
find	1,50	not much decisive	AV	0,00
use	1,50	not much decisive	AV	0,00
handle	1,50	not much decisive	AV	0,00
re-evaluate	1,00	can be decisive	AV	0,00
remember	2,00	not much decisive	AV	0,00
ignore	1,50	not much decisive	AV	0,00
assemble	1,50	not much decisive	AV	0,00
explore	1,50	not much decisive	AV	0,00
override	1,50	not much decisive	AV	0,00
look for	1,50	not much decisive	AV	0,00
break	1,50	not much decisive	AV	0,00
rewrite	1,50	not much decisive	AV	0,00
make	1,50	not much decisive	AV	0,00
avoid	1,50	not much decisive	AV	0,00
define	1,50	not much decisive	AV	0,00
decouple	1,50	not much decisive	AV	0,00
consume	1,50	not much decisive	AV	0,00

might be	2,00	not much decisive	OTV	0,00
execute	1,50	not much decisive	AV	0,00
test	1,50	not much decisive	AV	Test debt
review	2,00	not much decisive	AV	0,00
insert	1,50	not much decisive	AV	0,00
perform	1,50	not much decisive	AV	0,00
Eliminate	1,50	not much decisive	AV	0,00
check	1,00	can be decisive	AV	0,00
refactor	1,50	not much decisive	AV	Code debt
delete	1,50	not much decisive	AV	0,00
debug	1,50	not much decisive	AV	0,00
overlap	1,50	not much decisive	AV	Code debt
follow	1,50	not much decisive	AV	0,00
amend	1,50	not much decisive	AV	0,00
require	1,50	not much decisive	AV	0,00
Show	1,50	not much decisive	AV	0,00
maintain	1,50	not much decisive	AV	0,00
to do	1,50	not much decisive	AV	0,00
added	1,50	not much decisive	AV	0,00
modified	1,50	not much decisive	AV	0,00
verified	1,50	not much decisive	AV	0,00
tried	1,50	not much decisive	AV	0,00
copied	1,50	not much decisive	AV	0,00
rebuilt	1,50	not much decisive	AV	0,00
redid	1,50	not much decisive	AV	0,00
redone	1,50	not much decisive	AV	0,00
found	1,50	not much decisive	AV	0,00
overrode	1,50	not much decisive	AV	0,00
overridden	1,50	not much decisive	AV	0,00
broken	2,00	not much decisive	AV	0,00
broke	1,50	not much decisive	AV	0,00
rewrote	1,50	not much decisive	AV	0,00
rewritten	1,50	not much decisive	AV	0,00
made	1,50	not much decisive	AV	0,00
left	1,50	not much decisive	AV	0,00
shown	1,50	not much decisive	AV	0,00

Pattern	Score	Level	Combination	TD types
Blind	1,50	not much decisive	Adj	0,00
Drab	1,50	not much decisive	Adj	0,00
Nebulous	1,50	not much decisive	Adj	0,00
Tense	1,50	not much decisive	Adj	0,00
Hard	2,00	not much decisive	Adj	0,00
heavy	1,50	not much decisive	Adj	0,00
wrong	2,00	not much decisive	Adj	0,00
missing	2,00	not much decisive	Adj	0,00
inadequate	2,00	not much decisive	Adj	0,00
incomplete	2,00	not much decisive	Adj	0,00
unnecessary	2,00	not much decisive	Adj	0,00
slow	1,50	not much decisive	Adj	0,00
dangerous	2,00	not much decisive	Adj	0,00
erratic	2,00	not much decisive	Adj	0,00
inconsistent	2,00	not much decisive	Adj	0,00
obsolete	2,00	not much decisive	Adj	Code debt
unprepared	1,50	not much decisive	Adj	0,00
Unsafe	2,00	not much decisive	Adj	0,00
inefficient	2,00	not much decisive	Adj	0,00
undocumented	2,50	Decisive	Adj	Documentation debt
not friendly	2,00	not much decisive	Adj	0,00
not clear	2,00	not much decisive	Adj	0,00
large	1,50	not much decisive	Adj	0,00
dirty	2,00	not much decisive	Adj	0,00
not better	1,50	not much decisive	Adj	0,00
provisional	2,00	not much decisive	Adj	0,00

might be	2,00	not much decisive	OTV	0,00
temporary	4,00	Very decisive	Adj	Code debt, Design debt
deprecated	3,50	Very decisive	Adj	Code debt
tired	1,50	not much decisive	Adj	0,00
Critic	2,00	not much decisive	Adj	0,00
complicated	2,00	not much decisive	Adj	0,00
not correct	2,00	not much decisive	Adj	0,00
not stable	2,00	not much decisive	Adj	0,00
Strange	2,00	not much decisive	Adj	0,00
duplicate	2,50	Decisive	Adj	Code debt, Design debt
not right	2,00	not much decisive	Adj	0,00
Confuse	2,00	not much decisive	Adj	0,00
unwanted	2,00	not much decisive	Adj	0,00
bad	1,50	not much decisive	Adj	0,00
Long	1,50	not much decisive	Adj	0,00
uncommented	2,00	not much decisive	Adj	Documentation debt
Old	1,50	not much decisive	Adj	0,00
difficult	2,00	not much decisive	Adj	0,00
loose	1,50	not much decisive	Adj	0,00
far-fetched	1,50	not much decisive	Adj	0,00
messy	1,50	not much decisive	Adj	0,00
hairy	1,50	not much decisive	Adj	0,00
careful	1,50	not much decisive	Adj	0,00
impossible	1,50	not much decisive	Adj	0,00
not sufficient	2,00	not much decisive	Adj	0,00
weird	2,00	not much decisive	Adj	0,00
expensive	1,50	not much decisive	Adj	0,00
hurt	1,50	not much decisive	Adj	0,00
not reliable	1,50	not much decisive	Adj	Requirement debt
extra	1,50	not much decisive	Adj	0,00
dependent	1,50	not much decisive	Adj	Architecture debt
repeated	1,50	not much decisive	Adj	Code debt, Design debt
incorrect	2,00	not much decisive	Adj	0,00
fatal	2,00	not much decisive	Adj	0,00
uncool	1,50	not much decisive	Adj	0,00
unhappy	1,50	not much decisive	Adj	0,00
ugly	2,00	not much decisive	Adj	0,00
silly	1,50	not much decisive	Adj	0,00
senseless	2,00	not much decisive	Adj	0,00
stupid	2,00	not much decisive	Adj	0,00
retarded	1,50	not much decisive	Adj	0,00
unclean	2,00	not much decisive	Adj	0,00
redundant	2,00	not much decisive	Adj	0,00
no better	2,00	not much decisive	Adj	0,00
unnecessarily	2,00	not much decisive	Adj	0,00
Forgotten	3,00	Decisive	Adj	0,00
contradict	2,00	not much decisive	Adj	0,00
huge	1,50	not much decisive	Adj	0,00
complex	1,50	not much decisive	Adj	0,00
unhappy	1,50	not much decisive	Adj	0,00

Pattern	Score	Level	Combination	TD types
Currently	1,00	can be decisive	Adv	0,00
Now	1,50	not much decisive	Adv	0,00
later	2,00	not much decisive	Adv	0,00
any more	0,50	can be decisive	Adv	0,00
probably	0,50	can be decisive	Adv	0,00
before	1,00	can be decisive	Adv	0,00
correctly	1,00	can be decisive	Adv	0,00
incorrectly	1,50	not much decisive	Adv	0,00
less	1,00	can be decisive	Adv	0,00

might be	2,00	not much decisive	OTV	0,00
less efficient	3,00	Decisive	Adv	0,00
Here	1,00	can be decisive	Adv	0,00
perhaps	1,00	can be decisive	Adv	0,00
Yet	1,00	can be decisive	Adv	0,00
Really	1,00	can be decisive	Adv	0,00
not necessarily	1,50	not much decisive	Adv	0,00
directly	1,00	can be decisive	Adv	0,00
anymore	1,00	can be decisive	Adv	0,00
somewhere	1,50	not much decisive	Adv	0,00
someplace	1,50	not much decisive	Adv	0,00
instead	1,00	can be decisive	Adv	0,00
more	1,00	can be decisive	Adv	0,00
horribly	1,50	not much decisive	Adv	0,00
too	1,00	can be decisive	Adv	0,00
intentionally	1,00	can be decisive	Adv	0,00
fully	1,00	can be decisive	Adv	0,00
not enough	1,50	not much decisive	Adv	0,00
barely	1,50	not much decisive	Adv	0,00
Future	2,00	not much decisive	Adv	0,00
maybe	1,00	can be decisive	Adv	0,00
Never	1,00	can be decisive	Adv	0,00
Why	1,00	can be decisive	Adv	0,00
Why?	2,00	not much decisive	Adv	0,00

Pattern	Score	Level	Combination	TD types
:/	1,00	can be decisive	Symbol	0,00
:(	1,00	can be decisive	Symbol	0,00
:o	1,00	can be decisive	Symbol	0,00

Pattern	Score	Level	Combination	TD types
wtf	2,50	Decisive	Acronym	0,00
trap	4,00	Very decisive	Noun	0,00
ToDoItem	1,00	can be decisive	Abreviation	0,00
ToDoList	1,00	can be decisive	Abreviation	0,00
ASAP	1,00	can be decisive	Acronym	0,00
hardcode	2,00	not much decisive	Acronym	Code debt
hardcoded	2,00	not much decisive	Acronym	Code debt
kaboom	3,00	Decisive	Sound	0,00

Pattern	Score	Level	Combination	TD types
error	3,00	Decisive	Noun	Defect debt
mistake	3,00	Decisive	Noun	Defect debt
problem	2,00	not much decisive	Noun	0,00
problematic	2,00	not much decisive	Noun	0,00
fail	2,50	Decisive	Noun	Defect debt
discrepancy	2,00	not much decisive	Noun	0,00
Optimization	1,50	not much decisive	Noun	Code debt
workaround	2,50	Decisive	Noun	Code debt, Design debt
Bug	3,00	Decisive	Noun	Defect debt
limitation	2,00	not much decisive	Noun	0,00
Deadlock	2,50	Decisive	Noun	Code debt
defect	3,00	Decisive	Noun	Defect debt
issue	4,00	Very decisive	Noun	0,00
dependency	2,00	not much decisive	Noun	Architecture debt
clone	1,00	can be decisive	Noun	Code debt, Design debt
warning	3,00	Decisive	Noun	0,00
misunderstanding	2,50	Decisive	Noun	Documentation debt
bullshit	3,00	Decisive	Noun	0,00
kludge	2,50	Decisive	Noun	0,00
barf	3,00	Decisive	Noun	0,00

might be	2,00	not much decisive	OTV	0,00
yuck	3,00	Decisive	Noun	0,00

New Patterns				
will not be use			old file	
will not be delete			old code	
will not be reuse			expensive method	
would not be remove			extra design	
should not be maintain			extra goal	
should not be use			extra parameter	
should not be remove			interaction	
can be update			redundant data	
can be remove			required software	
can be found			required code	
can be redone			huge collection	
can be made			complex method	
can be use			complex design	
can be broken			can delete	
can be decouple			can execute	
can be replace			can change	
can be added			can read	
can be perform			can redo	
can be fix			can use	
can be overridden			can remove	
can be create			can show	
can be move			can make	
can be implement			can verify	
can be change			can create	
can be improve			can find	
can be refactor			can resolve	
can be execute			can decouple	
can be merged			can replace	
can be shown			can cause	
can be generate			can update	
may be use			can add	
may be added			can implement	
may be shown			can support	
may be overridden			can configure	
may be execute			can iterate	
may be remove			can handle	
may be split			can generate	
have to follow			shall implement	
have to create			shall show	
have to check			shall override	
have to implement			shall test	
have to store			shall replace	
have to find			shall add	
have to update			will add	
have to show			will try	
have to make			will test	
have to handle			will use	
must be define			will change	
must be added			will work	
must be overridden			will create	
must be remove			will make	
must be refactor			will generate	
must be use			will handle	
must be implement			will move	
must be test			will reuse	
must be handle			will jump	
needs to iterate			will resolve	

might be	2,00	not much decisive	OTV	0,00
needs to move			will check	
needs to use			will follow	
needs to implement			will override	
needs to show			will replace	
needs to change			will remove	
need to update			must handle	
need to remove			must find	
need to handle			must implement	
need to test			must generate	
need to create			must add	
need to add			must check	
need to find			must make	
need to support			must ignore	
need to use			must override	
need to override			must use	
need to modify			should update	
need to store			should remove	
need to implement			should work	
need to make			should generate	
should be move			should implement	
should be require			should show	
should be made			should create	
should be shown			should read	
should be ignore			should add	
should be found			should change	
should be insert			should check	
should be create			should replace	
should be delete			should override	
should be added			should handle	
should be remove			should move	
should be change			should amend	
should be improve			should split	
should be replace			should added	
should be use			should require	
should be update			would fix	
should be refactor			would work	
should be review			would add	
should be generate			would require	
should be execute			would cause	
should be rename			would support	
should be overridden			could find	
should be resolve			could check	
should be merged			could generate	
should be implement			could add	
should be store			could change	
should be rework			could implement	
should be split			could use	
shouldnt be create			could cause	
shouldnt be handle			could ignore	
shouldnt be found			could added	
shouldnt be require			could make	
should have check			could delete	
would be use			may remove	
would be handle			may show	
will be change			may change	
will be shown			may find	
will be added			may add	
will be create			may override	
will be use			may indicate	
will be solve			may generate	
will be ignore			may cause	
will be remove			may improve	
will be update			might cause	

might be	2,00	not much decisive	OTV	0,00
will be generate			might use	
will be delete			might check	
will be made			might work	
will be execute			might override	
will be check			might require	
will be store			hard coding	
will be move			wrong component	
will be modify			missing trigger	
will be found			inconsistent file	
could be use			temporary file	
could be made			deprecated method	
could be modify			deprecated class	
could be review			deprecated version	
should never change			critic network	
long class			critic subsystem	
old algorithm			critic base	
old list			critic analysis	
old project			critic class	
old version			duplicate code	
old implementation				

Pattern	Score	Level	Combination	TD types
Diagram	0	-	SE Noun	Documentation Debt
Activity Diagrams	0	-	SE Noun	Documentation Debt
Transition Activity	0	-	SE Noun	Documentation Debt
Interrupt-driven Models	0	-	SE Noun	Documentation Debt
Class Diagrams	0	-	SE Noun	Documentation debt
Layered Models	0	-	SE Noun	Documentation Debt
Diagram	0	-	SE Noun	Documentation Debt
Documentation	0	-	SE Noun	Documentation Debt
Specification	0	-	SE Noun	Documentation Debt
Model	0	-	SE Noun	Documentation Debt
Handbook	0	-	SE Noun	Documentation Debt
Sequence Diagrams	0	-	SE Noun	Documentation Debt
Metadata	0	-	SE Noun	Documentation Debt
Guidelines	0	-	SE Noun	Documentation Debt
Use case	0	-	SE Noun	Documentation Debt
Statechart Diagrams	0	-	SE Noun	Documentation Debt
Deployment Diagrams	0	-	SE Noun	Documentation Debt
Requirements Specification	0	-	SE Noun	Documentation Debt
Entity-relationship Diagrams	0	-	SE Noun	Documentation Debt
Software Design Notations	0	-	SE Noun	Documentation Debt
Collaborative Diagrams	0	-	SE Noun	Documentation Debt
Decision Tables And Diagrams	0	-	SE Noun	Documentation Debt
Flowcharts	0	-	SE Noun	Documentation Debt
Flowchart Component	0	-	SE Noun	Documentation Debt
Entity Attribute	0	-	SE Noun	Documentation Debt
Use Case Diagrams	0	-	SE Noun	Documentation debt
Information	0	-	SE Noun	Documentation Debt
Object Diagrams	0	-	SE Noun	Documentation Debt
Actor	0	-	SE Noun	Documentation Debt
Manager Models	0	-	SE Noun	Documentation Debt
Branch Transition	0	-	SE Noun	Architecture Debt
Architecture	0	-	SE Noun	Architecture Debt
System	0	-	SE Noun	Architecture Debt
COM	0	-	SE Noun	Architecture Debt
Polymorphism	0	-	SE Noun	Architecture Debt
Procedure	0	-	SE Noun	Architecture Debt
CORBA	0	-	SE Noun	Architecture Debt
Tier	0	-	SE Noun	Architecture Debt

might be	2,00	not much decisive	OTV	0,00
Hierarchy	0	-	SE Noun	Architecture Debt
Query	0	-	SE Noun	Architecture Debt
Peer To Peer Architecture	0	-	SE Noun	Architecture Debt
Multiprocessor Architectures	0	-	SE Noun	Architecture Debt
Portability	0	-	SE Noun	Architecture Debt
Peer	0	-	SE Noun	Architecture Debt
Architectural Design	0	-	SE Noun	Architecture Debt
Distributed Systems Architectures	0	-	SE Noun	Architecture Debt
Service-oriented System Architectur	0	-	SE Noun	Architecture Debt
Coupling	0	-	SE Noun	Architecture Debt
Aspect	0	-	SE Noun	Architecture Debt
Base	0	-	SE Noun	Architecture Debt
Stop Transition	0	-	SE Noun	Architecture Debt
Software Architecture	0	-	SE Noun	Architecture Debt
Application Architectures	0	-	SE Noun	Architecture Debt
Subsystem	0	-	SE Noun	Architecture Debt
Link	0	-	SE Noun	Architecture Debt
Module	0	-	SE Noun	Architecture Debt
Package	0	-	SE Noun	Architecture Debt
Component	0	-	SE Noun	Architecture Debt
Environment	0	-	SE Noun	Architecture Debt
Sql	0	-	SE Noun	Code Debt
Syntax	0	-	SE Noun	Code Debt
Code	0	-	SE Noun	Code Debt
Class	0	-	SE Noun	Code Debt
Class Attribute	0	-	SE Noun	Code Debt
Class Operation	0	-	SE Noun	Code Debt
Class Relationship	0	-	SE Noun	Code Debt
Deployment	0	-	SE Noun	Code Debt
Loop	0	-	SE Noun	Code Debt
Class Dependency	0	-	SE Noun	Code Debt
Large-grain Objects	0	-	SE Noun	Code Debt
Class Generalisation	0	-	SE Noun	Code Debt
Line	0	-	SE Noun	Code Debt
List	0	-	SE Noun	Code Debt
Parameter	0	-	SE Noun	Code Debt
Script	0	-	SE Noun	Code Debt
Object	0	-	SE Noun	Code Debt
Object Attribute	0	-	SE Noun	Code Debt
Class Strutural	0	-	SE Noun	Code Debt
Release	0	-	SE Noun	Code Debt
Class Agregation	0	-	SE Noun	Code Debt
Object-Class	0	-	SE Noun	Code Debt
Class Association	0	-	SE Noun	Code Debt
Class Corporation	0	-	SE Noun	Code Debt
Object Generalisation	0	-	SE Noun	Code Debt
Code-based Techniques	0	-	SE Noun	Code Debt
Interface	0	-	SE Noun	Code debt
Abstraction	0	-	SE Noun	Code Debt
Agility	0	-	SE Noun	Code Debt
Trigger	0	-	SE Noun	Code Debt
Array	0	-	SE Noun	Code Debt
Collection	0	-	SE Noun	Code Debt
File	0	-	SE Noun	Code Debt
Coding	0	-	SE Noun	Code Debt
Implementation	0	-	SE Noun	Code Debt
Compilers	0	-	SE Noun	Code Debt
Version	0	-	SE Noun	Code Debt
Commit	0	-	SE Noun	Code Debt
Function	0	-	SE Noun	Code Debt
Gui	0	-	SE Noun	Code Debt
Debuggers	0	-	SE Noun	Code Debt
Modification	0	-	SE Noun	Code Debt
Redundancy	0	-	SE Noun	Code Debt
Refactoring	0	-	SE Noun	Code Debt

might be	2,00	not much decisive	OTV	0,00
Reuse	0	-	SE Noun	Code Debt
Editors	0	-	SE Noun	Code Debt
listener	0	-	SE Noun	Code Debt
Source	0	-	SE Noun	Code Debt
Depuration	0	-	SE Noun	Code Debt
Method	0	-	SE Noun	Code Debt
Metric	0	-	SE Noun	Code Debt
Interpreters	0	-	SE Noun	Code Debt
Codification	0	-	SE Noun	Code Debt
Ide	0	-	SE Noun	Code Debt
Library	0	-	SE Noun	Code Debt
Algorithm	0	-	SE Noun	Code Debt
Cohesion	0	-	SE Noun	Code Debt
Compilation	0	-	SE Noun	Code Debt
Declaration	0	-	SE Noun	Code Debt
Logic	0	-	SE Noun	Code Debt
Resource Usages	0	-	SE Noun	Code Debt
Exception	0	-	SE Noun	Code Debt
Rollback	0	-	SE Noun	Code Debt
Control Structures	0	-	SE Noun	Code Debt
scrollbar	0	-	SE Noun	Code Debt
Routine	0	-	SE Noun	Code Debt
Language	0	-	SE Noun	Code Debt
Programming Language	0	-	SE Noun	Code Debt
Testing Team	0	-	SE Noun	People Debt
Implementation Team	0	-	SE Noun	People Debt
Analysis Team	0	-	SE Noun	People Debt
People	0	-	SE Noun	People Debt
Customers	0	-	SE Noun	People Debt
Software Engineers	0	-	SE Noun	People Debt
User	0	-	SE Noun	People Debt
Programmer	0	-	SE Noun	People Debt
Analyst	0	-	SE Noun	People Debt
Expertise	0	-	SE Noun	People Debt
Stakeholder	0	-	SE Noun	People Debt
Engineer	0	-	SE Noun	People Debt
Design Team	0	-	SE Noun	People Debt
Project	0	-	SE Noun	Design Debt
Reusable Units	0	-	SE Noun	Design debt
Object-oriented Decomposition	0	-	SE Noun	Design debt
Prototypes	0	-	SE Noun	Design debt
Pattern	0	-	SE Noun	Design Debt
Software Design	0	-	SE Noun	Design debt
Encapsulation	0	-	SE Noun	Design Debt
Data Flow Component	0	-	SE Noun	Design debt
Layer	0	-	SE Noun	Design Debt
Code	0	-	SE Noun	Design Debt
Design	0	-	SE Noun	Design Debt
Solution	0	-	SE Noun	Design Debt
Object	0	-	SE Noun	Design Debt
Object Dependency	0	-	SE Noun	Design debt
Framework	0	-	SE Noun	Infrastructure Debt
Data Process	0	-	SE Noun	Infrastructure Debt
Program	0	-	SE Noun	Infrastructure Debt
Data Store	0	-	SE Noun	Infrastructure Debt
Data	0	-	SE Noun	Infrastructure Debt
Fork Transition	0	-	SE Noun	Infrastructure Debt
Software	0	-	SE Noun	Infrastructure Debt
Join Transition	0	-	SE Noun	Infrastructure Debt
Special Transition	0	-	SE Noun	Infrastructure Debt
Start Transition	0	-	SE Noun	Infrastructure Debt
Systems Organisation	0	-	SE Noun	Infrastructure Debt
Function-oriented Pipelining	0	-	SE Noun	Infrastructure Debt
Concurrent Transition	0	-	SE Noun	Infrastructure Debt
Database	0	-	SE Noun	Infrastructure Debt

might be	2,00	not much decisive	OTV	0,00
Client-server Models	0	-	SE Noun	Infrastructure Debt
Memory	0	-	SE Noun	Infrastructure Debt
External Entity	0	-	SE Noun	Infrastructure Debt
Distributed Client-server Architecture	0	-	SE Noun	Infrastructure Debt
Backup	0	-	SE Noun	Infrastructure Debt
Product	0	-	SE Noun	Infrastructure Debt
Protocol	0	-	SE Noun	Infrastructure Debt
Proxy	0	-	SE Noun	Infrastructure Debt
Server	0	-	SE Noun	Infrastructure Debt
Channel	0	-	SE Noun	Infrastructure Debt
Hd	0	-	SE Noun	Infrastructure Debt
Communication	0	-	SE Noun	Infrastructure Debt
Processor	0	-	SE Noun	Infrastructure Debt
Table	0	-	SE Noun	Infrastructure Debt
Data Input	0	-	SE Noun	Infrastructure Debt
Data Output	0	-	SE Noun	Infrastructure Debt
Disk	0	-	SE Noun	Infrastructure Debt
Site	0	-	SE Noun	Infrastructure Debt
Technology	0	-	SE Noun	Infrastructure Debt
Storage	0	-	SE Noun	Infrastructure Debt
Transaction	0	-	SE Noun	Infrastructure Debt
Backup	0	-	SE Noun	Infrastructure Debt
Channel	0	-	SE Noun	Infrastructure Debt
Report	0	-	SE Noun	Infrastructure Debt
Firewall	0	-	SE Noun	Infrastructure Debt
Hardware	0	-	SE Noun	Infrastructure Debt
Infrastructure	0	-	SE Noun	Infrastructure Debt
Screen	0	-	SE Noun	Infrastructure Debt
Structure	0	-	SE Noun	Infrastructure Debt
Dictionary	0	-	SE Noun	Infrastructure Debt
Distributed Client	0	-	SE Noun	Infrastructure Debt
Network	0	-	SE Noun	Infrastructure Debt
Distributed Network	0	-	SE Noun	Infrastructure Debt
Distributed Server	0	-	SE Noun	Infrastructure Debt
Support	0	-	SE Noun	Infrastructure Debt
Event Process	0	-	SE Noun	Process Debt
Development	0	-	SE Noun	Process Debt
Processing Step	0	-	SE Noun	Process Debt
Maintenance	0	-	SE Noun	Process Debt
Process	0	-	SE Noun	Process Debt
Paradigm	0	-	SE Noun	Process Debt
Effort	0	-	SE Noun	Process Debt
Technique	0	-	SE Noun	Process Debt
Estimate	0	-	SE Noun	Process Debt
Interviews	0	-	SE Noun	Process Debt
Effectiveness	0	-	SE Noun	Process Debt
Timeline	0	-	SE Noun	Process Debt
Security	0	-	SE Noun	Process Debt
Analysis	0	-	SE Noun	Process Debt
Allocation	0	-	SE Noun	Process Debt
Homologation	0	-	SE Noun	Process Debt
Measurement	0	-	SE Noun	Process Debt
Installation	0	-	SE Noun	Process Debt
Interaction	0	-	SE Noun	Process Debt
Cost	0	-	SE Noun	Process Debt
Methodology	0	-	SE Noun	Process Debt
Distribution	0	-	SE Noun	Process Debt
Transaction-processing Systems	0	-	SE Noun	Process Debt
Managed Process	0	-	SE Noun	Process Debt
Rule	0	-	SE Noun	Process Debt
Template	0	-	SE Noun	Process Debt
Schedule	0	-	SE Noun	Process Debt
Management	0	-	SE Noun	Process Debt
Reutilization	0	-	SE Noun	Process Debt
Goal	0	-	SE Noun	Process Debt

might be	2,00	not much decisive	OTV	0,00
Business	0	-	SE Noun	Process Debt
Domain-specific Services	0	-	SE Noun	Service Debt
Service	0	-	SE Noun	Service Debt
Service	0	-	SE Noun	Service Debt
Service Provider	0	-	SE Noun	Service Debt
Client	0	-	SE Noun	Service Debt
Service Registry	0	-	SE Noun	Service Debt
Service Requestor	0	-	SE Noun	Service Debt
Distribution Process	0	-	SE Noun	Service Debt
Interrupt	0	-	SE Noun	Defect Debt
Defect Tracking	0	-	SE Noun	Defect Debt
Failure	0	-	SE Noun	Defect Debt
Risk	0	-	SE Noun	Defect Debt
Fault	0	-	SE Noun	Defect Debt
Debug	0	-	SE Noun	Defect Debt
Tolerance	0	-	SE Noun	Defect Debt
Bug	0	-	SE Noun	Defect Debt
Traceability	0	-	SE Noun	Defect Debt
Defect Identification	0	-	SE Noun	Defect Debt
Software Requirements	0	-	SE Noun	Requirement Debt
Requirements	0	-	SE Noun	Requirement Debt
Emergent Properties	0	-	SE Noun	Requirement Debt
Functional Requirements	0	-	SE Noun	Requirement Debt
Non-functional Requirements	0	-	SE Noun	Requirement Debt
Delivery Requirements	0	-	SE Noun	Requirement Debt
Ethical Requirements	0	-	SE Noun	Requirement Debt
Implementation Requirements	0	-	SE Noun	Requirement Debt
Scope	0	-	SE Noun	Requirement Debt
Performance Requirements	0	-	SE Noun	Requirement Debt
Portability Requirements	0	-	SE Noun	Requirement Debt
Privacy Requirements	0	-	SE Noun	Requirement Debt
Reliability Requirements	0	-	SE Noun	Requirement Debt
Safety Requirements	0	-	SE Noun	Requirement Debt
Space Requirements	0	-	SE Noun	Requirement Debt
Standards Requirements	0	-	SE Noun	Requirement Debt
Usability Requirements	0	-	SE Noun	Requirement Debt
Process Requirements	0	-	SE Noun	Requirement Debt
Artifact	0	-	SE Noun	Requirement Debt
Product Requirements	0	-	SE Noun	Requirement Debt
Quantifiable Requirements	0	-	SE Noun	Requirement Debt
System Requirements	0	-	SE Noun	Requirement Debt
Users Requirements	0	-	SE Noun	Requirement Debt
Requirements Analysis	0	-	SE Noun	Requirement Debt
Requirements Elicitation	0	-	SE Noun	Requirement Debt
Beta-test	0	-	SE Noun	Test Debt
Software Testing	0	-	SE Noun	Test Debt
Test Activities	0	-	SE Noun	Test Debt
Test-case Generation	0	-	SE Noun	Test Debt
Test-case	0	-	SE Noun	Test Debt
Test Environment Development	0	-	SE Noun	Test Debt
Test Execution	0	-	SE Noun	Test Debt
Test Log	0	-	SE Noun	Test Debt
Constrution Testing	0	-	SE Noun	Test Debt
Test Planning	0	-	SE Noun	Test Debt
Monitoring	0	-	SE Noun	Test Debt
Test Results Evaluation	0	-	SE Noun	Test Debt
Integration Testing	0	-	SE Noun	Test Debt
System Testing	0	-	SE Noun	Test Debt
Unit Testing	0	-	SE Noun	Test Debt
Test Techniques	0	-	SE Noun	Test Debt

might be	2,00	not much decisive	OTV	0,00
GUI Testing	0	-	SE Noun	Test Debt
Object-oriented Testing	0	-	SE Noun	Test Debt
Protocol Conformance Testing	0	-	SE Noun	Test Debt
Real-time Systems Testing	0	-	SE Noun	Test Debt
Safety-critical Systems Testing	0	-	SE Noun	Test Debt
Web-based Testing	0	-	SE Noun	Test Debt
Mutation Testing	0	-	SE Noun	Test Debt
Adhoc Testing	0	-	SE Noun	Test Debt
Testing Issues	0	-	SE Noun	Test Debt
Exploratory Testing	0	-	SE Noun	Test Debt
Random Testing	0	-	SE Noun	Test Debt
Robustness Testing	0	-	SE Noun	Test Debt
Limitations Of testing	0	-	SE Noun	Test Debt
Test Criteria	0	-	SE Noun	Test Debt
Testabability	0	-	SE Noun	Test Debt
Test	0	-	SE Noun	Test Debt
Stress Testing	0	-	SE Noun	Test Debt
Usability Testing	0	-	SE Noun	Test Debt
Software Constrution	0	-	SE Noun	Build Debt
Speed	0	-	SE Noun	Build Debt
Performance	0	-	SE Noun	Build Debt

BIBLIOGRAPHY

1 FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, v. 39, n. 11, p. 27–34, 1996. ISSN 00010782.

2 MALDONADO, E.; SHIHAB, E.; TSANTALIS, N. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering*, p. 1–1, 2017. ISSN 0098-5589.

3 FALESSI, D.; SHAW, M. a.; SHULL, F.; MULLEN, K.; STEIN, M. Practical Considerations , Challenges , and Requirements of Tool-Support for Managing Technical Debt. p. 16–19, 2013.

4 AVGERIOU, P.; KRUCHTEN, P.; OZKAYA, I.; SEAMAN, C.; SEAMAN, C. Managing Technical Debt in Software Engineering Edited by. *Dagstuhl Reports*, v. 6, n. 4, p. 110–138, 2016.

5 IZURIETA, C.; VETRÒ, A.; ZAZWORKA, N.; CAI, Y.; SEAMAN, C.; SHULL, F. Organizing the technical debt landscape. *3rd International Workshop on Managing Technical Debt (MTD) - Proceedings*, p. 23–26, 2012.

6 ALVES, N. S. R.; RIBEIRO, L. F.; CAIRES, V.; MENDES, T. S.; SPÍNOLA, R. O. Towards an Ontology of Terms on Technical Debt. In: *Sixth International Workshop on Managing Technical Debt (MTD)*. [S.l.: s.n.], 2014. p. 1–7. ISBN 9781479967919.

7 SEAMAN, C.; GUO, Y.; ZAZWORKA, N.; SHULL, F.; IZURIETA, C.; CAI, Y.; VETRÒ, A. Using technical debt data in decision making: Potential decision approaches. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, p. 45–48, 2012.

8 BOHNET, J.; DÖLLNER, J. Monitoring code quality and development activity by software maps. *Proceeding of the 2nd working on Managing technical debt - MTD '11*, p. 9, 2011. ISSN 02705257.

9 GUO, Y.; SPÍNOLA, R. O.; SEAMAN, C. Exploring the Costs of Technical Debt Management – A Case Study. *Empirical Software Engineering*, v. 1, p. 1–24, 2014. ISSN 1382-3256.

10 TOM, E.; AURUM, A.; VIDGEN, R. An exploration of technical debt. *Journal of Systems and Software*, v. 86, n. 6, p. 1498–1516, 2013. ISSN 01641212.

11 HUANG, Q.; SHIHAB, E.; XIA, X.; LO, D.; LI, S. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 2017. ISSN 1382-3256.

12 LI, Z.; AVGERIOU, P.; LIANG, P. A Systematic Mapping Study on Technical Debt and its Management. *Journal of Systems and Software*, Elsevier Ltd., v. 101, p. 193–220, 2014. ISSN 01641212.

13 ALVES, N. S.; MENDES, T. S.; MENDONÇA, M. G. de; SPÍNOLA, R. O.; SHULL, F.; SEAMAN, C. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, v. 70, p. 100–121, 2016. ISSN 09505849.

14 MENDES, T. S.; ALMEIDA, D. A.; ALVES, N. S. R.; SPÍNOLA, R. O.; MENDONÇA, M. VisMinerTD - An Open Source Tool to Support the Monitoring of the Technical Debt Evolution using Software Visualization. In: *17th International Conference on Enterprise Information Systems*. [S.l.: s.n.], 2015.

15 Lanza, Michelle; Marinescu, R. *Object-Oriented Metrics in Practics*. [S.l.: s.n.], 2006. ISBN 9783540244295.

16 MAALEJ, W.; HAPPEL, H.-J. Can Development Work Describe Itself? In: *7th IEEE Working Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2010. p. 191–200. ISBN 978-1-4244-6802-7.

17 STEIDL, D.; HUMMEL, B.; JUERGENS, E. Quality Analysis of Source Code Comments. In: *21st International Conference on Program Comprehension (ICPC)*. [S.l.]: Ieee, 2013. p. 83–92. ISBN 978-1-4673-3092-3.

18 CRUZES, D. S.; DYBÅ, T. Synthesizing evidence in software engineering research. In: *IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM*. [S.l.: s.n.], 2010. p. 1. ISBN 9781450300391.

19 CAMPBELL, D. T.; FISKE, D. W. Convergent and discriminant validation by the multitrait-multimethod matrix. *Psychological Bulletin*, v. 56, n. 2, p. 81–105, 1959. ISSN 0033-2909.

20 SHULL, F.; SINGER, J.; SJOBERG, D. *Guide to Advanced Empirical Software Engineering*. [S.l.]: Springer, 2008. ISBN 9781848000438.

21 JICK, T. D. Mixing Qualitative and Quantitative Methods : Triangulation in Action Mixing. *Qualitative Methodology*, v. 24, n. 4, p. 602–611, 1979.

22 FARIAS, M. A. F.; NOVAIS, R.; COLAÇO, M.; CARVALHO, L. P. d. S.; MENDONÇA, M.; SPÍNOLA, R. O. A Systematic Mapping Study on Mining Software Repositories. *31st ACM/SIGAPP Symposium on Applied Computing*, 2016.

23 BASILI, V.; SHULL, F.; LANUBILE, F. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, v. 25, n. 4, p. 456–473, 1999. ISSN 0098-5589.

24 FARIAS, M. A. F.; SILVA, A. B.; MENDONÇA, M. G. de; SPÍNOLA, R. O. A Contextualized Vocabulary Model for Identifying Technical Debt on Code Comments. *7th International Workshop on Managing Technical Debt*, n. ii, p. 25–32, 2015.

25 FARIAS, M. A. F.; SILVA, A. B.; MENDONÇA, M. G. de; SPÍNOLA, R. O.; KALINOWSKI, M. Investigating the Use of a Contextualized Vocabulary in the Identification of Technical Debt : A Controlled Experiment. *18Th International Conference on Enterprise Information System (ICEIS)*, v. 1, p. 369–378, 2016.

26 FARIAS, M. A. F.; SANTOS, J. A.; KALINOWSKI, M.; MENDONÇA, M.; SPÍNOLA, R. Investigating the Identification of Technical Debt through Code Comment Analysis. In: *Enterprise Information Systems*. [S.l.]: Springer International Publishing, 2017. cap. 14, p. 284–309. ISBN 978-3-319-62385-6.

27 FARIAS, M. A. F.; SPíNOLA, R.; MENDONçA, M. FindTD I, Mendeley Data. v. 1, 2017. Available from Internet: ⟨http://dx.doi.org/10.17632/ndx9zddmf6.1⟩.

28 FARIAS, M. A. F.; SPíNOLA, R.; MENDONçA, M. FindTD II, Mendeley Data. v. 1, 2017. Available from Internet: ⟨http://dx.doi.org/10.17632/yf3c2xrs8h.3⟩.

29 FARIAS, M. A. F.; SPíNOLA, R.; MENDONçA, M. FindTD III, Mendeley Data. v. 1, 2017. Available from Internet: ⟨http://dx.doi.org/10.17632/x7gfnxk3m6.4⟩.

30 FARIAS, M. A. F.; SPíNOLA, R.; MENDONçA, M. FindTD IV, Mendeley Data. v. 1, 2017. Available from Internet: ⟨http://dx.doi.org/10.17632/xbw6d7vbn4.3⟩.

31 CUNNINGHAM, W. The WyCash portfolio management system. p. 29–30, 1992.

32 BROWN, N.; OZKAYA, I.; SANGWAN, R.; SEAMAN, C.; SULLIVAN, K.; ZAZWORKA, N.; CAI, Y.; GUO, Y.; KAZMAN, R.; KIM, M.; KRUCHTEN, P.; LIM, E.; MACCORMACK, A.; NORD, R. Managing technical debt in software-reliant systems. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*. [S.l.: s.n.], 2010. p. 47. ISBN 9781450304276.

33 THEODOROPOULOS, T.; HOFBERG, M.; KERN, D. Technical debt from the stakeholder perspective. *Proceeding of the 2nd working on Managing technical debt - MTD '11*, p. 43, 2011. ISSN 02705257.

34 KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. *IEEE Software*, v. 29, n. 6, p. 18–21, 2012. ISSN 07407459.

35 ZAZWORKA, N.; SPÍNOLA, R. O.; VETRO', A.; SHULL, F.; SEAMAN, C. A Case Study on Effectively Identifying Technical Debt. In: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering - EASE '13*. New York, New York, USA: ACM Press, 2013. p. 42–47. ISBN 9781450318488.

36 ERNST, N. A.; BELLOMO, S.; OZKAYA, I.; NORD, R. L.; GORTON, I. Measure it? Manage it? Ignore it? software practitioners and technical debt. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE*. [S.l.: s.n.], 2015. p. 50–60. ISBN 9781450336758.

37 RIBEIRO, L. F.; FARIAS, M. A. D. F.; MENDONÇA, M.; OLIVEIRA, R. Decision Criteria for the Payment of Technical Debt in Software Projects: A Systematic Mapping Study. *In Proceedings of the 18th International Conference on Enterprise Information Systems (ICEIS)*, v. 1, p. 572–579, 2016.

38 NUGROHO, A.; VISSER, J.; KUIPERS, T. An Empirical Model of Technical Debt and Interest Software Improvement Group. *Proceeding of the 2nd working on Managing technical debt MTD 11*, p. 1, 2011.

39 ZAZWORKA, N.; ACKERMANN, C. CodeVizard: A Tool to Aid the Analysis of Software Evolution. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, v. 2, n. 4, p. 63:1—-63:1, 2010.

40 VISA, A. Technology of Text Mining. In: *Proceedings of Machine Learning and Data Mining in Pattern Recognition, Second International Workshop, MLDM*. [S.l.: s.n.], 2001. p. 10–20. ISBN 3540423591. ISSN 16113349.

41 COLAÇO, M.; MENDONÇA, M.; ANDRÉ, M.; FARIAS, D. F.; HENRIQUE, P. A Neurolinguistic-based Methodology for Identifying OSS Developers Context-Specific Preferred Representational Systems. In: *Context*. [S.l.: s.n.], 2012. p. 112–121. ISBN 9781612082301.

42 HEMMATI, H.; NADI, S.; BAYSAL, O.; KONONENKO, O.; WANG, W.; HOLMES, R.; GODFREY, M. W. The MSR Cookbook: Mining a decade of research. *2013 10th Working Conference on Mining Software Repositories (MSR)*, Ieee, p. 343–352, may 2013.

43 KONCHADY, M. *Text Mining Application Programming*. [S.l.]: Charles River Media, 2006. ISBN 1-58450-460-9.

44 OGADA, K.; MWANGI, W.; CHERUIYOT, W. N-gram Based Text Categorization Method for Improved Data Mining. v. 5, n. 8, p. 35–44, 2015.

45 COIS, C. A.; KAZMAN, R. Natural language processing to quantify security effort in the software development lifecycle. In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*. [S.l.: s.n.], 2015. v. 2015-Janua, p. 716–721. ISBN 1891706373. ISSN 23259000.

46 DEMEYER, S.; MURGIA, A.; WYCKMANS, K.; LAMKANFI, A. Happy birthday! A trend analysis on past MSR papers. *IEEE International Working Conference on Mining Software Repositories*, p. 353–362, 2013. ISSN 21601852.

47 GRAOVAC, J.; KOVAČEVIĆ, J.; PAVLOVIĆ-LAŽETIĆ, G. Language Independent n-Gram-Based Text Categorization with Weighting Factors : A Case Study. *JIDM - Journal of Information and Data Management*, v. 6, n. 1, p. 4–17, 2015.

48 HOWDEN, W. E. Comments Analysis and Programming Errors. v. 16, n. 1, 1990.

49 GEGICK, M.; ROTELLA, P.; XIE, T. Identifying security bug reports via text mining: An industrial case study. *Proceedings - International Conference on Software Engineering*, p. 11–20, 2010. ISSN 02705257.

50 LAMKANFI, A.; DEMEYER, S.; GIGER, E.; GOETHALS, B. Predicting the severity of a reported bug. *Proceedings - International Conference on Software Engineering*, p. 1–10, 2010. ISSN 02705257.

51 BAJAJ, K.; PATTABIRAMAN, K.; MESBAH, A. Mining questions asked by web developers. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, p. 112–121, 2014.

52 LICORISH, S. A.; MACDONELL, S. G. Combining text mining and visualization techniques to study teams' behavioral processes. In: *4th IEEE Workshop on Mining Unstructured Data (MUD)*. [S.l.: s.n.], 2014. p. 16–20. ISBN 9781479967933.

53 FARIAS, M. A. F.; ORTINS, P.; NOVAIS, R.; COLAC, M.; MENDONC, M. Recovering Valuable Information Behaviour from OSS Contributors : An Exploratory Study. *The 26th International Conference on Software Engineering & Knowledge Engineering*, p. 474–477, 2014.

54 FARIAS, M.; NOVAIS, R.; ORTINS, P.; COLAÇO, M.; MENDONÇAA, M. Analyzing Distributions of Emails and Commits from OSS Contributors through Mining Software Repositories : An Exploratory Study. *In Proceedings of the 17th International Conference on Enterprise Information Systems*, p. 303–310, 2015.

55 HASSAN, E. The road ahead for Mining Software Repositories. *Frontiers of Software Maintenance, 2008. FoSM 2008.*, p. 48–57, 2008.

56 ROBLES, G. Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories Proceedings. In: *Proceedings - International Conference on Software Engineering*. [S.l.: s.n.], 2010. p. 171–180. ISBN 9781424468034. ISSN 02705257.

57 NAGUIB, H.; NARAYAN, N.; BRÜGGE, B.; HELAL, D. Bug report assignee recommendation using activity profiles. *IEEE International Working Conference on Mining Software Repositories*, p. 22–30, 2013. ISSN 21601852.

58 VALDIVIA, H. G.; SHIHAB, E. Characterizing and predicting blocking bugs in open source projects. In: *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. [S.l.: s.n.], 2014. p. 72–81. ISBN 9781450328630.

59 ZHANG, F.; MOCKUS, A.; KEIVANLOO, I.; ZOU, Y. Towards building a universal defect prediction model. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, p. 182–191, 2014.

60 STEIDL, D.; HUMMEL, B.; JUERGENS, E. Incremental origin analysis of source code files. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, p. 42–51, 2014.

61 SAHA, R. K.; ROY, C. K.; SCHNEIDER, K. a.; PERRY, D. E. Understanding the evolution of type-3 clones: An exploratory study. *IEEE International Working Conference on Mining Software Repositories*, p. 139–148, 2013. ISSN 21601852.

62 GIL, J.; GOLDSTEIN, M.; MOSHKOVICH, D. An empirical investigation of changes in some software properties over time. *IEEE International Working Conference on Mining Software Repositories*, p. 227–236, 2012. ISSN 21601852.

63 FARIAS, M. A. F.; SPíNOLA RODRIGO, N. R.; MENDONçA, M. MSR Mapping Packge, Mendeley Data. v. 1, 2015. Available from Internet: ⟨http://dx.doi.org/10.17632/vsf24gh4k8.1⟩.

64 KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. *A survey and taxonomy of approaches for mining software repositories in the context of software evolution*. 2007.

65 STOREY, M.-a.; RYALL, J.; BULL, R. I.; MYERS, D.; SINGER, J. TODO or To Bug : Exploring How Task Annotations Play a Role in the Work Practices of Software Developers. In: *International Conference on Software Engineering(ICSE)*. [S.l.: s.n.], 2008. p. 251–260. ISBN 9781605580791.

66 SOUZA, S. C. B.; ANQUETIL, N.; OLIVEIRA, K. M.; de SouzaNicolas AnquetilKáthia M. de Oliveira, S. C. B. Which documentation for software maintenance? *Journal of the Brazilian Computer Society*, v. 12, n. 3, p. 31–44, 2006. ISSN 0104-6500.

67 SHOKRIPOUR, R.; ANVIK, J.; KASIRUN, Z. M.; ZAMANI, S. Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation. *IEEE International Working Conference on Mining Software Repositories*, p. 2–11, 2013. ISSN 21601852.

68 FREITAS, J. L.; Da Cruz, D.; HENRIQUES, P. R. A comment analysis approach for program comprehension. *Proceedings of the IEEE 35th Software Engineering Workshop*, p. 11–20, 2012. ISSN 1550-6215.

69 FLURI, B.; WÜRSCH, M.; GALL, H. C. Do code and comments co-evolve? On the relation between source code and comment changes. In: *Proceedings - Working Conference on Reverse Engineering, WCRE*. [S.l.]: IEEE, 2007. ISBN 0769530346. ISSN 10951350.

70 ETZKORN, L. H.; DAVIS, C. G.; BOWEN, L. L. The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics*, v. 33, n. 11, p. 1731–1756, 2001. ISSN 03782166.

71 HOWARD, M. J.; GUPTA, S.; POLLOCK, L.; VIJAY-SHANKER, K. Automatically mining software-based, semantically-similar words from comment-code mappings. *2013 10th Working Conference on Mining Software Repositories (MSR)*, Ieee, p. 377–386, may 2013.

72 YANG, J.; TAN, L. Inferring semantically related words from software context. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. [S.l.]: IEEE, 2012. p. 161–170. ISBN 978-1-4673-1761-0.

73 GUPTA, S.; MALIK, S.; POLLOCK, L.; VIJAY-SHANKER, K. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. *IEEE International Conference on Program Comprehension*, p. 3–12, 2013. ISSN 1063-6897.

74 SALVIULO, F.; MATEMATICA, D.; BASILICATA, U.; SCANNIELLO, G. Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance : Results from an Ethnographically-informed Study with Students and Professionals. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering. ACM*. [S.l.: s.n.], 2014. p. 48. ISBN 9781450324762.

75 POTDAR, A.; SHIHAB, E. An Exploratory Study on Self-Admitted Technical Debt. In: *IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. p. 91–100. ISBN 978-1-4799-6146-7.

76 MALDONADO, E. S.; SHIHAB, E. Detecting and Quantifying Different Types of Self-Admitted Technical Debt. In: *7th International Workshop on Managing Technical Debt*. [S.l.: s.n.], 2015. p. 9–15.

77 SILVA, M. C. O.; VALENTE, M. T.; TERRA, R. Does Technical Debt Lead to the Rejection of Pull Requests? n. ii, 2016.

78 MALETIC, J. I.; COLLARD, M. L.; MARCUS, A. Source Code Files as Structured Documents. In: *Proceedings. 10th International Workshop on*. [S.l.: s.n.], 2002. p. 289–292.

79 WEHAIBI, S.; SHIHAB, E.; GUERROUJ, L. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 1, p. 179–188. ISBN 978-1-5090-1855-0.

80 BAVOTA, G.; RUSSO, B. A large-scale empirical study on self-admitted technical debt. In: *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. [S.l.: s.n.], 2016. p. 315–326. ISBN 9781450341868.

81 WONGTHONGTHAM, P.; CHANG, E.; DILLON, T.; SOMMERVILLE, I. Software Engineering Ontology - the Instance Knowledge (Part I). *International Journal of Computer Science and Network Security USA*, v. 7, n. 2, p. 15–26, 2007.

82 WONGTHONGTHAM, P.; CHANG, E.; DILLON, T.; SOMMERVILLE, I. Development of a software engineering ontology for multisite software development. *IEEE Transactions on Knowledge and Data Engineering*, v. 21, n. 8, p. 1205–1217, 2009. ISSN 10414347.

83 FARIAS, M. A. F.; SPíNOLA, R.; MENDONçA, M. eXcomment, Mendeley Data. v. 4, 2016. Available from Internet: ⟨http://dx.doi.org/10.17632/s43m4t6t6b.4⟩.

84 WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in Software Engineering*. [S.l.: s.n.], 2012. ISBN 9783642290435.

85 LEMOS, O. a. L.; DE, A. C.; ZANICHELLI, F. C.; LOPES, C. V. Thesaurus-Based Automatic Query Expansion for Interface-Driven Code Search Categories and Subject Descriptors. In: *11th Working Conference on Mining Software Repositories - MSR*. [S.l.: s.n.], 2014. p. 212–221. ISBN 9781450328630.

86 HOST, M.; WOHLIN, C.; THELIN, T. Experimental context classification: incentives and experience of subjects. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, p. 470–478, 2005.

87 SALMAN, I.; MISIRLI, A. T.; JURISTO, N. *Are Students Representatives of Profess...s in Software Engineering Experiments_.pdf*. 2015, 666–676 p.

88 SANTOS, J. A.; MENDONÇA, M. G. de; SANTOS, C. P.; NOVAIS, R. L. The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development*, v. 2, n. 1, p. 11, 2014.

89 FINN, R. *A Note on Estimating the Reliability of Categorical Data*. [S.l.]: Educational and Psychological Measurement, 1970. 71–76 p. ISSN 0013-1644. ISBN 0013-1644.

90 LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *Biometrics*, v. 33, n. 1, p. 159–174, 1977. ISSN 0006-341X.

91 J. Cohen. *Statistical power analysis for the behavioral sciences*. 2 edition. ed. [S.l.: s.n.], 1988.

92 SNEDECOR, G. W.; COCHRAN, W. G. *Statistical Methods. Ames*. [S.l.]: Iowa State University Press Iowa, 1967.

93 SPÍNOLA, R.; ZAZWORKA, N.; SEAMAN, C.; SHULL, F. Investigating Technical Debt Folklore. *5th International Workshop on Managing Technical Debt*, p. 1–7, 2013.

94 WOHLIN, C.; RUNESON, P.; HÖST, M. *Experimentation in Software Engineering: an introduction*. [S.l.]: Kluwer Academic Publishers Norwell, 2000. ISBN 0-7923-8682-5.

95 SULLIVAN, G. M.; ARTINO, A. R. Analyzing and Interpreting Data From Likert-Type Scales. *Journal of Graduate Medical Education*, v. 5, n. 4, p. 541–542, 2013. ISSN 1949-8349.

96 ELLSBERG, M.; HEISE, L. *Researching Violence Against Women. A PRACTICAL GUIDE FOR RESEARCHERS AND ACTIVISTS*. Washington: World Health, 2005. ISBN 92 4 154647 6.

97 MELOROSE, J.; PERROY, R.; CAREAS, S. *Experimentation in Software Engineering*. [S.l.: s.n.], 2015. ISSN 1098-6596. ISBN 9788578110796.

98 BACHMANN, A.; BERNSTEIN, A. When process data quality affects the number of bugs: Correlations in software engineering datasets. *Proceedings - International Conference on Software Engineering*, p. 62–71, 2010. ISSN 02705257.

99 MALDONADO, E. S.; SHIHAB, E. Detecting and Quantifying Different Types of Self-Admitted Technical Debt. In: *7th International Workshop on Managing Technical Debt*. [S.l.: s.n.], 2015. p. 9–15. ISBN 9781467373784.

100 KITCHENHAM, B.; Pearl Brereton, O.; BUDGEN, D.; TURNER, M.; BAILEY, J.; LINKMAN, S. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology*, Elsevier B.V., v. 51, n. 1, p. 7–15, jan 2009. ISSN 09505849.

101 Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M. Systematic mapping studies in software engineering. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. [S.l.: s.n.], 2008. p. 68–77. ISBN 0-7695-2555-5. ISSN 02181940.

102 BRERETON, P.; KITCHENHAM, B. a.; BUDGEN, D.; TURNER, M.; KHALIL, M. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, v. 80, n. 4, p. 571–583, apr 2007. ISSN 01641212.

103 WOHLIN, C.; RUNESON, P.; da Mota Silveira Neto, P. A.; ENGSTRÖM, E.; do Carmo Machado, I.; ALMEIDA, E. S. de. On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, Elsevier Inc., v. 86, n. 10, p. 2594–2610, oct 2013. ISSN 01641212.

104 KITCHENHAM, B. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2007.

105 NOVAIS, R. L.; TORRES, A.; MENDES, T. S.; MENDONÇA, M.; ZAZWORKA, N. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, Elsevier B.V., v. 55, n. 11, p. 1860–1883, nov 2013. ISSN 09505849.

106 FARIAS, M.; COLAÇO, M.; SPÍNOLA, R. O.; NETO, M. G. D. M. Identifying Technical Debt through Code Comment Analysis. *Doctoral Consortium on Enterprise Information Systems*, n. Dceis, p. 9–14, 2016.

107 PUNTER, T.; CIOLKOWSKI, M.; FREIMUT, B.; JOHN, I. Conducting on-line surveys in software engineering. *International Symposium on Empirical Software Engineering. ISESE. Proceedings.*, 2003.

108 RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, v. 14, n. 2, p. 131–164, 2009. ISSN 13823256.

109 SJØBERG, D. I. K.; HANNAY, J. E.; HANSEN, O.; KAMPENES, V. B.; KARAHASANOVIĆ, A.; LIBORG, N. K.; REKDAL, A. C. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, v. 31, n. 9, p. 733–753, 2005. ISSN 00985589.

110 FERNANDEZ, A.; INSFRAN, E.; ABRAHÃO, S. Usability evaluation methods for the web: A systematic mapping study. *Information and Software Technology*, Elsevier B.V., v. 53, n. 8, p. 789–817, aug 2011. ISSN 09505849.

111 ROTELLA, P.; CHULANI, S. Implementing quality metrics and goals at the corporate level. *Proceedings - International Conference on Software Engineering*, p. 113–122, 2011. ISSN 02705257.

112 LOTUFO, R.; PASSOS, L.; CZARNECKI, K. Towards improving bug tracking systems with game mechanisms. *IEEE International Working Conference on Mining Software Repositories*, p. 2–11, 2012. ISSN 21601852.

113 XIA, X.; LO, D.; WANG, X.; ZHOU, B. Tag recommendation in software information sites. *IEEE International Working Conference on Mining Software Repositories*, p. 287–296, 2013. ISSN 21601852.

114 STEVENS, R.; GANZ, J.; FILKOV, V.; DEVANBU, P.; CHEN, H. Asking for (and about) permissions used by android apps. *IEEE International Working Conference on Mining Software Repositories*, p. 31–40, 2013. ISSN 21601852.

115 PINTO, G.; CASTOR, F.; LIU, Y. D. Mining questions about software energy consumption. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, p. 22–31, 2014.

116 PONZANELLI, L.; BAVOTA, G.; Di Penta, M.; OLIVETO, R.; LANZA, M. Mining StackOverflow to turn the IDE into a self-confident programming prompter. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, p. 102–111, 2014.

117 NUGROHO, A.; CHAUDRON, M. R. V.; ARISHOLM, E. Assessing UML design metrics for predicting fault-prone classes in a Java system. *Proceedings - International Conference on Software Engineering*, p. 21–30, 2010. ISSN 02705257.

118 RAO, S.; KAK, A. Retrieval from Software Libraries for Bug Localization : A Comparative Study of Generic and Composite Text Models. *Work*, p. 43–52, 2011. ISSN 02705257.

119 KARUS, S.; GALL, H. A Study of Language Usage Evolution in Open Source Software. p. 13–22, 2011. ISSN 02705257.

120 MURGIA, A.; TOURANI, P.; ADAMS, B.; ORTU, M. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press, New York, New York, USA, p. 262–271, 2014.

121 NUSSBAUM, L.; ZACCHIROLI, S. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. *Proceedings - International Conference on Software Engineering*, p. 52–61, 2010. ISSN 02705257.

122 FARIAS, M. A. F.; SANTOS, J. A.; KALINOWSKI, M.; MENDONÇA, M.; SPÍNOLA, R. Investigating the Identification of Technical Debt through Code Comment Analysis. In: *Enterprise Information Systems*. [S.l.]: Springer International Publishing, 2017. cap. 14, p. 1–26. ISBN 978-3-319-62385-6.

123 PASSOS A. F.O., F. M. N. C. M. M. S. R. Taxonomy and data extracted from Systematic Mapping Study on Software Comments Analysis, Mendeley Data. v. 3, 2017. Available from Internet: ⟨http://dx.doi.org/10.17632/xhrk7f4w3p.3⟩.

124 WIERINGA, R.; MAIDEN, N.; MEAD, N.; ROLLAND, C. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. *Requirements Engineering*, v. 11, n. 1, p. 102–107, 2006. ISSN 09473602.