Classes-Chave em Sistemas Orientados a Objetos: Detecção e Uso

Liliane do Nascimento Vale

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia - MG
2017

Key Classes in Object-Oriented Systems: Detection and Use

Liliane do Nascimento Vale

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia - MG
2017

Liliane do Nascimento Vale

Classes-Chave em Sistemas Orientados a Objetos: Detecção e Uso

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Engenharia de Software

Supervisor: Marcelo de Almeida Maia

Uberlândia - MG
2017

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO – FACOM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO – PPGCO

The undersigned hereby certify they have read and recommend to the PPGCO for acceptance the doctoral thesis entitled **"Key Classes in Object-Oriented Systems: Detection and Use"** submitted by **"Liliane do Nascimento Vale"** as part of the requirements for obtaining the **Doctor's degree in Computer Science**.

Uberlândia, 13 de novembro de 2017

Supervisor: Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Examining Committee Members:

Prof. Dr. Marco Túlio Valente
Universidade Federal de Minas Gerais

Prof. Dr. Uirá Kulesza
Universidade Federal do Rio Grande do Norte

Prof. Dr. Flávio de Oliveira Silva
Universidade Federal de Uberlândia

Prof. Dr. Fabiano Azevedo Dorça
Universidade Federal de Uberlândia

Ao meu filho, que mal te sinto mas, mesmo assim, já te quero e te amo tanto que nem sei como explicar...

Acknowledgements

A Deus em primeiro lugar, pelas pessoas que ele colocou em meu caminho. Algumas delas me inspiram, me ajudam, me desafiam e me encorajam a ser cada dia melhor. Em especial agradeço...

A meus pais Milson e Esmeralda que me amam incondicionalmente.

Ao meu esposo Leonardo pela paciência que teve durante esta etapa, pelo carinho e amor que sempre dedicou a mim, me fortalecendo nos momentos difíceis e proporcionando grandes alegrias.

Aos meus amigos e colegas por terem me apoiado. Em especial, a Humberto Lidio, simplesmente não tenho palavras para expressar minha gratidão. A Carlos Eduardo, Henrique, Núbia e Tércio, em auxiliar nos experimentos.

E com imensa gratidão agradeço ao professor Marcelo Maia pela orientação segura e competente, seus estímulos constantes e testemunhos de seriedade, permitiram-me concretizar este estudo. Agradeço também pela compreensão de meus limites, auxiliando-me com sua imensa sabedoria de forma imprescindível para o desenvolvimento deste trabalho. Foram valiosas suas contribuições para o meu crescimento intelectual e pessoal.

Finalmente, agradeço aos participantes, que contribuiram com a pesquisa e a FAPEG pelo auxílio financeiro.

"Adoro Reticências... Aqueles três pontos intermitentes que insistem em dizer que nada está fechado, que nada acabou, que algo sempre está por vir! A vida se faz assim! Nada pronto, nada definido. Tudo sempre em construção. Tudo ainda por se dizer... Nascendo... Brotando... Sublimando... Vivo assim... Numa eterna reticência... Para que colocar ponto final? O que seria de nós sem a expectativa de continuação?" (Nilson Furtado)

Resumo

Vários sistemas orientados a objetos, tais como Lucene, Tomcat, Javac tem seus respectivos projetos (*designs*) documentados usando classes-chave, definidas como sendo classes importantes/centrais para compreender o projeto de sistemas orientados a objetos. Considerando este fato, e considerando que geralmente a arquitetura não é formalmente documentada para auxiliar os desenvolvedores a entenderem e avaliarem o projeto do software, é proposta Keecle, uma abordagem baseada em análise dinâmica e estática para detecção de classes-chave de maneira semi-automática. É proposta a aplicação de mecanismos de filtragem no espaço de busca dos dados dinâmicos, para obter um conjunto reduzido de classes-chave. A abordagem é avaliada com quatorze sistemas de código aberto e proprietários, a fim de verificar se as classes encontradas correspondem às classes-chave definidas na documentação ou definidas pelos desenvolvedores. Os resultados foram analisados em termos de precisão e *recall* e são superiores às abordagens da literatura. O papel das classes-chave para avaliar o projeto também foi investigado. Foi avaliado se a organização das classes-chave em um grafo de dependências, o qual destaca relações de dependência explícitas no código fonte, é um mecanismo adequado para avaliar o *design*. Foi analisado estatisticamente, se classes-chave são mais propensas a bad smells, e se tipos específicos de *bad smells* estão associados a diferentes níveis de métricas de coesão e acoplamento. Além disso, a *propriedade* (*ownership*) das classes-chave foi analisada, indicando concentração em um conjunto reduzido de desenvolvedores. Por fim, foram conduzidos um estudo experimental com estudantes e um survey com desenvolvedores para avaliar a documentação baseada em classes-chave. Os resultados demonstram que a documentação baseada em classes-chave apresenta resultados que indicam a viabilidade de uso como documentação complementar à existente ou como documentação principal em ambientes onde a documentação não está disponível.

Palavras-chave: engenharia reversa, classes-chave, *design*, *smells*, experimento, análise dinâmica.

Abstract

Several object-oriented systems, such as Lucene, Tomcat, Javac have their respective design documented using key-classes, defined as important/central classes to understand the object-oriented design. Considering this fact, and considering that, in general, software architecture is not formally documented to help developers understanding and assessing software design, Keecle is proposed as an approach based on dynamic and static analysis for detection of key classes in a semi-automatic way. The application of filtering mechanisms on the search space of the dynamic data is proposed in order to obtain a reduced set of key classes. The approach is evaluated with fourteen proprietary and open source systems in order to verify that the found classes correspond to the key classes of the ground-truth, which is defined from the documentation or defined by the developers. The results were analyzed in terms of precision and recall, and have shown to be superior to the state-of-the-art approach. The role of key classes in assessing design has also been investigated. The organization of the key classes in a dependency graph, which highlights explicit dependency relations in the source code, was evaluated to be adequate for design comprehension and assessment. Key classes were evaluated whether they are more prone to bad smells, and whether specific types of bad smells are associated with different levels of cohesion and coupling metrics. In addition, the ownership of key classes was shown to be more concentrated in a reduced set of developers. Finally, we conducted an experimental study with students and a survey with developers to evaluate documentation based on key classes. The results indicate that the documentation based on key classes are a feasible alternative for use as complementary documentation to the existing one, or for use as main documentation in environments where documentation is not available.

Keywords: reverse engineering, key-classes, design, smells, dynamic analysis, experiment.

List of Figures

List of Tables

List of Algorithm

Contents

CHAPTER 1

Introduction

Software evolution is especially important during the system development process. In general, software systems constantly change to meet new requirements, to fix bugs, to optimize source code, to integrate new features, etc.

In this context, program comprehension has an important role during software maintenance. In order to timely change applications with quality, developers need to understand the design and the current implementation as well. Understanding is facilitated when the developer is the owner or an expert of the respective system, or when there is adequate supporting documentation.

However, due to the pressure on developers to deliver new software releases quickly and with low cost, documentation is, in general, neither available nor updated. In this context, the opportunity for the use of reverse engineering techniques is open. Reverse engineering is an alternative to study source code, when there is no other source of reliable information. There are two techniques to perform reverse engineering: static analysis and dynamic analysis. Static analysis can provide a complete description of the system, because it can be applied to the complete source code of a program. However, it does not capture important behavioral events for understanding the software architecture because execution scenarios of the application are not considered (CORNELISSEN et al., 2009). On the other hand, dynamic analysis relies on the system properties captured during its execution. Dynamic data, often in the form of execution traces, is collected using strategies that configure scenarios related to only those parts of interest for the analysis. Execution traces capture the actual behavior of the system and can have a tree-based structure, that can be used in software design understanding strategies (CORNELISSEN et al., 2009). Several works have used dynamic analysis to recover architectural views (WALKER et al., 2000), identify design patterns (HEUZEROTH et al., 2003a), features (EISENBARTH; KOSCHKE; SIMON, 2003) (GREEVY; DUCASSE, 2005) and architectural styles (YAN et al., 2004). These approaches had to deal with the challenges related to the trace size to prevent significant effort from developers when analyzing and understanding the trace data.

In the context of reverse engineering, software architecture reconstruction plays an important role. In general, software architecture is documented in a package based structure, because it is easier to map architectural components to actual artifacts. However, quite often this is not the best architectural organization (GARCIA et al., 2013). Moreover when the architectural documentation is available, it is often outdated because of phenonema, such as architectural drift or erosion (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). To alleviate these problems, several architecture recovery techniques have already been proposed, nevertheless there are still problems that hinder the use of those techniques (DUCASSE; POLLET, 2009a)(SARTIPI, 2003)(ASTUDILLO; VALDES; BECERRA, 2012).

A recent study performed a comparative analysis to measure the accuracy of the recovery techniques use by six different architectures by use of eight ground-truth architectures, and this study indicated that the limitations concerning these techniques are related to accuracy, to the conditions under which techniques succeed or fail, to the number and size of selected systems, etc., (GARCIA et al., 2013). For instance, the average accuracy using the MoJoFM measure was 45% (WEN; TZERPOS, 2004). An apparently successful approach that combined dynamic and static analysis for software clustering showed an MojoFM accuracy of 87.83% (PATEL; HAMOU-LHADJ; RILLING, 2009). However, this approach was evaluated only with the Weka[1] sofware and most of the retrieved components consisted of classes from the same package, which may not be a general representative of software architectures, as reported in (GARCIA et al., 2013). Dynamic analysis has been used with static analysis to provide relevant information of behavioral aspects during the software architecture reconstruction.

Cornelissen et al, (2009) analyzed 176 articles related to dynamic analysis applied to different areas of software engineering, such as feature location, bug detection, architectural reconstruction, etc. In that study, 13 articles used dynamic analysis for software architecture recovery. Summing up, large architectural components extraction from the source code is complex, and still suffers from a low accuracy of performance (GARCIA; IVKOVIC; MEDVIDOVIC, 2013a). The available tools require significant developer effort to understand the retrieved information, limiting the use of such tools.

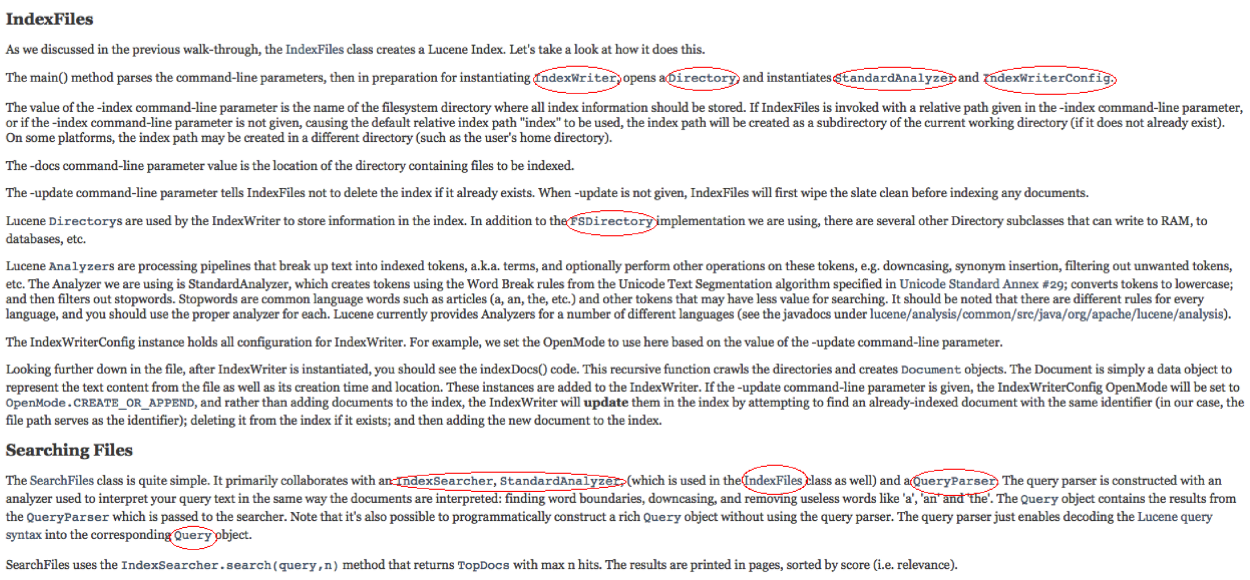
A recent study has highlighted the importance of producing documentation containing architectural description on open-source projects and emphasizes the main problems found in the current documentation (ROBILLARD; MEDVIDOVIC, 2016). This work highlighted a case study involving the analysis of architectural documentation of 18 source code softwares. Each invited contributor re-documented the architecture of a system on a limited number of pages and adopted their own criteria for producing the document. Subsequently, the authors of that paper reviewed the documentations and concluded that there was no uniform criterion for documenting a software application. So, this contributes for creating a gap between the creators and consumers because of the manual nature of its creation (ROBILLARD et al., 2017).

[1] http://www.cs.waikato.ac.nz/ml/weka/documentation.html

Alternatively, we have observed that several real-world systems such as such as Lucene[2], Tomcat[3] and Javac[4] use some few classes to document its architectural design, an example is shown on Figure 1. In this way, an alternative would be a documentation based on key classes. In this context, **key classes** are defined as being the core classes to comprehend the object-oriented design. So, a documentation based on key classes would be an alternative to substitute or complement design documentation.

As a motivating example to show the importance of a design documentation based on key classes to understand and assess design can be taken from the documentation of Lucene[5], shown in Figure 1.

Among the classes referenced in that documentation, an interesting one is the *IndexWriter* class. This class has the worst rates for cohesion and coupling metrics considered: RFC, LCOM, CBO and DIT (CHIDAMBER; KEMERER, 1994). The RFC metric indicated that this class has 458 points of execution, with a lack of cohesion (LCOM metric) equals to 9414, and the coupling between object classes (CBO metric) equals to 48. The values were recovered using cOPE tool (KAKARONTZAS et al., 2013) and indicate that *IndexWriter* is a complex class and has the occurrence of smells. Five kinds of smells were detected: *AntiSingleton, ClassDataShouldBePrivate, ComplexClass, LongParameterList* and *SpaghettiCode*.



IndexFiles

As we discussed in the previous walk-through, the IndexFiles class creates a Lucene Index. Let's take a look at how it does this.

The main() method parses the command-line parameters, then in preparation for instantiating IndexWriter, opens a Directory and instantiates StandardAnalyzer and IndexWriterConfig.

The value of the -index command-line parameter is the name of the filesystem directory where all index information should be stored. If IndexFiles is invoked with a relative path given in the -index command-line parameter, or if the -index command-line parameter is not given, causing the default relative index path "index" to be used, the index path will be created as a subdirectory of the current working directory (if it does not already exist). On some platforms, the index path may be created in a different directory (such as the user's home directory).

The -docs command-line parameter value is the location of the directory containing files to be indexed.

The -update command-line parameter tells IndexFiles not to delete the index if it already exists. When -update is not given, IndexFiles will first wipe the slate clean before indexing any documents.

Lucene Directorys are used by the IndexWriter to store information in the index. In addition to the FSDirectory implementation we are using, there are several other Directory subclasses that can write to RAM, to databases, etc.

Lucene Analyzers are processing pipelines that break up text into indexed tokens, a.k.a. terms, and optionally perform other operations on these tokens, e.g. downcasing, synonym insertion, filtering out unwanted tokens, etc. The Analyzer we are using is StandardAnalyzer, which creates tokens using the Word Break rules from the Unicode Text Segmentation algorithm specified in Unicode Standard Annex #29; converts tokens to lowercase; and then filters out stopwords. Stopwords are common language words such as articles (a, an, the, etc.) and other tokens that may have less value for searching. It should be noted that there are different rules for every language, and you should use the proper analyzer for each. Lucene currently provides Analyzers for a number of different languages (see the javadocs under lucene/analysis/common/src/java/org/apache/lucene/analysis).

The IndexWriterConfig instance holds all configuration for IndexWriter. For example, we set the OpenMode to use here based on the value of the -update command-line parameter.

Looking further down in the file, after IndexWriter is instantiated, you should see the indexDocs() code. This recursive function crawls the directories and creates Document objects. The Document is simply a data object to represent the text content from the file as well as its creation time and location. These instances are added to the IndexWriter. If the -update command-line parameter is given, the IndexWriterConfig OpenMode will be set to OpenMode.CREATE_OR_APPEND, and rather than adding documents to the index, the IndexWriter will **update** them in the index by attempting to find an already-indexed document with the same identifier (in our case, the file path serves as the identifier); deleting it from the index if it exists; and then adding the new document to the index.

Searching Files

The SearchFiles class is quite simple. It primarily collaborates with an IndexSearcher, StandardAnalyzer, (which is used in the IndexFiles class as well) and a QueryParser. The query parser is constructed with an analyzer used to interpret your query text in the same way the documents are interpreted: finding word boundaries, downcasing, and removing useless words like 'a', 'an' and 'the'. The Query object contains the results from the QueryParser which is passed to the searcher. Note that it's also possible to programmatically construct a rich Query object without using the query parser. The query parser just enables decoding the Lucene query syntax into the corresponding Query object.

SearchFiles uses the IndexSearcher.search(query,n) method that returns TopDocs with max n hits. The results are printed in pages, sorted by score (i.e. relevance).

Figure 1 – Lucene Overview using a Documentation based on Key Classes.

To better understand the problem with *IndexWriter*, we located 65 open issues[6] associated with design problems involving this class, and noticed that all of these have an *Unresolved* status

[2] https://lucene.apache.org/core/6_5_1/demo/overview-summary.html
[3] https://tomcat.apache.org/tomcat-5.5-doc/architecture/overview.html
[4] http://openjdk.java.net/groups/compiler/doc/compilation-overview/
[5] https://lucene.apache.org/core/
[6] https://issues.apache.org/jira/issues/?jql=project%20%3D%20LUCENE

since 2009. The discussions' threads on those issues demonstrate the concern of the developers with this class indicating the need for refactoring.

So, *IndexWriter* is a key class for understanding and assessing design. However, we also can not neglect the possibility of non-key classes with similar design problems. A non-key class may also have structural design problems indicated by software metrics, but because non-key classes are, by definition, not critical to understand and assess design, they would not require higher priority. Nonetheless, because of the controlling nature of the key classes, they are expected to be more prone to present structural problems compared to non-key classes.

The advantage of knowing the key classes of an application, allows the developer to get a concrete overview of the system organization. Because key classes are likely to be directly related to design, if there are any design problems in these classes, those problems are likely to be more critical. This enables the developer to perform a design assessment focused on those classes and possibly, point out new design decisions during maintenance activities.

According to this example on *IndexWriter*, our objective is to use a reduced set of key classes to understand how to use the information conveyed in those structural and social properties to improve architectural knowledge and design assessment.

In order to find automatically the key classes in a system, there is already an approach proposed by Zaidman and Demeyer (ZAIDMAN; DEMEYER, 2008) to identify the most important classes in a system - the *key classes*. They characterized the key classes as typically possessing a lot of "control" within the application. In order to find these "controller classes", they presented a detection approach that is based on dynamic coupling and webmining, obtaining precision of around 50%. Other recent approaches have been proposed (DING; LI; HE, 2016), (MEYER; SIY; BHOWMICK, 2014) and (SORA, 2015). Moreover, these authors did not report on concrete evidence that the awareness of them is a useful information for developers, leaving a gap for further investigation.

So, in this thesis, instead of trying to improve the current techniques for recovering architectural components for design understanding and assessment, we build on the idea that several architectural documentation are organized around the description of few classes. Thus, we propose, Keecle[7], a semi-automatic way for finding *key classes* considered as important classes to understand and assess the design in object-oriented systems. It is intended to be an alternative way to provide architectural knowledge, where the concepts of the key classes would be likely mapped to those that are central to comprehend the software architecture.

For finding and evaluating key classes, our approach combines dynamic and static analyses. Dynamic analysis is used to capture and filter execution traces in order to find the key classes. Static analysis is used to provide more evidence that key classes, especially those recovered by Keecle, are an important means to understand and to assess software design.

[7] Kee has the same sound of "key" and "Cle" is a contracted sound for "cl"as

Newcomer contributors face several barriers between the time they decide to engage a new project and the time their first contribution is accepted (STEINMACHER et al., 2016). Two important classes of identified barriers are documentation problems and technical hurdles. The first is related to, among others, the outdated or non-existent documentation, and the second is related to, among others, code/architecture hurdles, which include bad code/design and cognitive problems during program/design comprehension. In this context, we propose also a documentation based on key classes to aid newcomers.

1.1 Objectives and Contributions

Considering our motivation for finding key classes with potential properties to comprehend and assess design, this work aims at proposing and evaluating a technique that extracts key classes, which supposedly give an initial understanding of the software design regarding structural and ownership properties. This objective can be organized in more specific objectives that would together achieve the overall goal of our work as follows:

- to propose a novel technique to identify key classes of a software system that can be provided for developers as a high-level overview to help understand important structure and relations of the software;
- to provide an empirical evaluation of the technique using open source and proprietary systems, aiming at outperforming the state-of-art techniques.
- to organize the key classes into a high-level overview that could help in a supplementary documentation. The goal is to investigate whether dependency graphs produces a degree of adherence with the documentation. This degree of adherence can benefit developers in cases where the software documentation is not available, or it complementing current documentation.
- to analyze the presence of specific bad smells in key classes and if there is any relationship with the cohesion and coupling metrics.
- to evaluate the ownership pattern on key classes. The goal is to understand the notion of responsibility of the developers on key classes. Finally, we evaluate the frequency of commits to define the level of ownership and analyze their relationship to key classes.
- to evaluate the role of semi-automatically detected key classes for understanding design. Experimental study with human subjects are aimed to evaluate quantitatively and qualitatively the value added by key classes on the comprehension of software design.

1.2 Hypotheses

Considering the motivation and objectives, we formulated a set of hypothesis related to technique to find the key classes, to their organization in terms of dependency graphs, to the study regarding social and structural properties and whether a documentation based on key classes can complement/replace a traditional documentation and overall help developers during the design assessment. These hypotheses are described following.

- **H1)** *A reduced set of key classes can be obtained from reverse engineering techniques using dynamic analysis.*

  Argumentation: The key classes presented important properties in previous studies which showed that such classes have strong control over the application (ZAIDMAN; DEMEYER, 2008). When we consider an execution trace tree, supposedly those key classes should be at the highest levels of the tree. If these nodes are at the highest levels of the tree, have stronger control over the application, as all other method calls will be controlled by those upper level nodes. So, classes in those upper level nodes would have a higher chance of being a key class. This hypothesis will be verified in Chapter 3.

- **H2)** *Key classes organized in a dependency graph is a strategy that complements the available documentation, showing important dependency relationships, and it also support undocumented environments.*

  Argumentation: The dependency graph structure may reveal a distinct reality compared to the actual documentation. In general, human-written documentation shows a simplified situation that does not necessarily match source code. The dependency graph of the key classes can display undesirable dependencies. On the dependency graph, can occur any dependency (cyclic dependency) that breaks this rule violating of the structure of the system. These dependencies are not always avoidable, so warnings may help developers to get them under control. This hypothesis will be investigate in Chapter 4.

- **H3)** *Key classes are more prone to low cohesion with high coupling, and this fact can be associated with the high occurrence of bad smells on key classes in relation to non-key classes.*

  Argumentation: Key classes are intrinsically related to design, as they have a strong control over the software. This situation would be more likely to influence the quality of the code. So, we investigate if there is any association of source code to the occurrence of bad smells. Moreover, to understand how key classes may impact design quality, we investigate if classical indicators for assessing modularity (coupling and cohesion) have distinct levels in key classes when compared to non-key classes, hence, investigate the relation between coupling and cohesion indicators and the occurrence of smells. This hypothesis will be investigated in Chapter 4.

- **H4)** *The ownership pattern of the key classes is different when compared to non-key classes.*

  Argumentation: The key classes are important classes of the system as already mentioned and therefore the distribution of developers work on those classes in relation to other classes would be more likely distinct compared to non-key classes. This hypothesis will be investigated in Chapter 4.

- **H5)** *Design documentation based on key classes can complement existing documentation or be a replacement for it.*

  Argumentation: The set of key classes highlights classes that are important from the design viewpoint and therefore may serve as basis for the representation a general organization of the system. Because documentation based on key classes is produced using dynamic analysis, providing a straight relation to the actual behavior of the software would benefit cognitive activities, and therefore would benefit more accurate solutions during comprehension activities. This hypothesis will be investigated in Chapter 5.

- **H6)** *A documentation based on key classes helps newcomers to understand an application.*

  Argumentation: Documentation based on key classes is more likely to be simple and straightforward, because the set of key classes may be chosen to be small. The rationale is that a small set of key classes can guide the analysis of the design more quickly rather than navigating on all available source files, in case when documentation is not available. This hypothesis will be investigated in Chapter 5.

1.3 Thesis Outline

Thesis statement:

The set of key classes detected by Keecle is an adequate source of information for producing documentation to effectively help developers to understand and assess design.

The structure of this thesis is organized in the following chapters:

- Chapter 2 provides basic concepts of architecture recovery, reverse engineering, program comprehension, bad smells, metrics of software that assist in the proposed solution.
- Chapter 3 presents Keecle, a semi-automatic proposal for the recovery of keys classes. Initially, an overview of Keecle approach using dynamic analysis is presented, describing how execution traces are captured, compressed, transformed into more compact subtrees, and also how the key classes are mined from those subtrees. Following on, the study settings to evaluate the accuracy regarding the Keecle approach and the evaluation results. Finally, a discussion is made considering threats to validity.

- Chapter 4 we present the study setting for analyzing properties of key classes in a structural and social context and present the results concerning those properties considering static analysis:
  - A studied property is the likelihood of key classes association to bad smells. First, we analyze the proneness of the occurrence of bad smells (BROWN et al., 1998) in key classes compared to the rest of the classes. Also, we analyze if the occurrence of specific bad smells are associated to different levels in cohesion and coupling metrics.
  - We propose a mechanism to organize key classes in a dependency graph to explicitly complement and visualize undesired dependencies, since they can affect the structure of the project, these class dependencies are typically neither documented nor complete. We performed a comparative study to analyze the degree of adherence between produced output and actual documentation focusing on circular dependencies to assess design.
  - Another studied property is related to the ownership of key class, and thus has a social context. We evaluated the distribution of key classes among developers to understand how ownership compares to non-key classes.
- Chapter 5 presents the experimental evaluation of the proposed approach, experimental design, results, discussions and conclusions. Two studies were conducted to evaluate quantitatively and qualitatively the value added by key classes on the comprehension of software design.
  - In the first study, students (potential newcomers in Open Source Systems - OSS) were surveyed in order to evaluate the useful of key classes as a starting point for comprehending an application.
  - In the second study, expert developers were surveyed in order to evaluate the role of key classes and whether a documentation based on key classes can complement or replace a traditional documentation.
- Chapter 6 presents related work, highlighting state of the art based on this thesis.
- Chapter 7 presents the conclusion of this study and proposed future work.

1.4 Publications

From this thesis, we have published the following work:

- Vale, L. N. and Maia, M. A. Keecle: Mining key architecturally relevant classes using dynamic analysis. Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. Pages 566–570. ERA Track.

- Vale, L. N. and Maia, M. A. On the Properties of Design-Relevant Classes for Design Anomaly Assessment. Program Comprehension (ICPC), 2017 IEEE/ACM International Conference on. Pages 332-335. ERA Track.

CHAPTER 2

Background

In this chapter, we present fundamental concepts related to this thesis. We introduce concepts of architecture recovery (Section 2.1). Next, a presentation is given of reverse engineering concepts in particular for dynamic and static analyses (Section 2.2). In Section 2.3, we present concepts concerning program comprehension, since our approach retrieves data as initial knowledge of the software architecture. In Section 2.4 Naïve Bayes technique is presented because it is classification model used in the algorithm that select key classes. In sequence, in Section 2.5, measures traditional such as recall and precision are presented to calculate the accuracy of this approach. Following this, in Section 2.6 we introduce concepts on bad smells, because we will analyze the prevalence of smells in key classes. Finally, metrics from (CHIDAMBER; KEMERER, 1994) are presented for evaluating the complexity of the key classes.

2.1 Architecture Recovery

Architecture emphasizes the global organization of the system, and distinct definitions are given to software architecture in the literature. Among them, we highlight two related to our object of study: *Architecture is a set of principal design decisions about a software system* (TAYLOR; MEDVIDOVIĊ; DASHOFY, 2009). The software architecture of a program or computing system is the structure or structures of the system, which comprise of software elements, the externally visible properties of those elements, and the relationships among them (BASS; CLEMENTS; KAZMAN, 1998). Software architecture is a very important topic due to the understanding, analysis, reusability, evolution and management of legacy systems.

Large organizations have in general a significant base of legacy systems. These systems represent a high development effort over a long period, and as such bring with them a wealth of knowledge about the business, which often can not be obtained from any other source of information available in the organization. Understanding these systems and their structural organization have been the constant concern of software engineers. According to (KAZMAN; CARRIÈRE, 1999) the development of software rarely begins from zero. It is usually restricted

by compatibility, or the use of legacy systems. In this case, it is necessary to find alternative techniques to retrieve relevant information from legacy systems such as architecture recovery .

Architecture recovery is a process by which higher levels of abstraction are identified and extracted from existing software systems (DUCASSE; POLLET, 2009b). Architecture recovery and reengineering to handle legacy code is critical for large and complex systems. Architecture recovery deals with the issues of recovering the past design decisions that have been taken by the experts during the development of a system. These are decisions that have been lost due to some reasons: they have never been documented, or when they were documented they were not frequently revised.

In order to support the software architecture recovery, various techniques, methods and tools have been proposed in the literature. A recent study (DUCASSE; POLLET, 2009b) presented a state of the art of software architecture reconstruction approaches. Reverse engineering is commonly used in these situations and therefore it is described in the next section.

2.2 Reverse Engineering

Reverse engineering is an essential technique in the architecture reconstruction process, as it enables the understanding of the system through the identification of components and its relationships, creating abstractions from this information (MÜLLER et al., 2000).

The software system code is the source of information that is most accessible, reliable and available when other artifacts are missing or out of date. In this case, reverse engineering is a process of examination and understanding software, to recapture or recreate the design and understand the requirements currently implemented by the software, presenting them in a higher level of abstraction (CHIKOFSKY; CROSS, 1990).

The information is extracted from the source code, helping us to understand the system (e.g., dependency relationships) and to find out specific problems in the system (e.g., violation of rules, duplicated code, smells, complexity of the code, etc.). Reverse engineering tools deal primarily with two tasks. The first task is to analyze source code and extract an abstract model from the source, whereas the second is to carry out some exploratory operations in this abstract model.

There are variations in the strategies concerning reverse engineering: static and dynamic analyses both used that will be detailed in the next subsections.

2.2.1 Dynamic Analysis

Dynamic analysis is used to extract representations that reflect system behavior at runtime. These representations consist of traces that are event logs generated by the program execution.

This technique has the potential to provide a precise view of a software system because it displays the actual behavior of the system.

The information collected at runtime facilitates for the understanding of dynamic architectural views of the system. Jerding and Rugaber reported on a study which claims that the dynamic models are essential for understanding architecture (JERDING; RUGABER, 1997). In other words, the understanding of a system architecture requires the identification of its components and the means by which interaction between such occurs in order to achieve the goals. The information may have details ranging from classes to architectural high-level views. Among the benefits of dynamic analysis are the availability of information, and, in the context of object-oriented software, exposure of the identities of objects. Data capture in a system execution occurs through interpretation (e.g., using the Java virtual machine) or instrumentation, these data collected during the dynamic analysis of the system are named execution traces.

We used a tool to capture execution traces proposed by Sobreira and Maia (2008). In this paper a visual tool to analyze the intersection of feature elements and source code elements from different matrix perspectives was proposed. To identify where the specified features are located in the execution trace, the developer must inform during the execution of the scenario when a feature starts and when it ends. They developed an instrumentation tool that asks the developer to inform a label to mark the beginning a new feature, immediately before triggering the scenario activity corresponding to a feature. When the execution of that feature ends, the developer must inform such event. This process has to be repeated until the developer executes all planned scenario. The result is an execution trace file for each thread started within the execution of the whole scenario. Each line of each trace file describes a method call completely qualified and its respective *timestamp* indicating when the method has started. The complete qualification of the method call is important to understand which class and which package has participated in the execution of each feature. The captured data have properties that make it possible to analyze it for various purposes as pointed out by Cornelissen et al. (2009).

However, one problem faced by dynamic analysis is the volume of the events extracted during software execution. In general, the data tend to be very large due to the existence of loops and recursion, making handling and analysis difficult. In order to contribute to the solution of this problem, one can considered the techniques of (HAMOU-LHADJ; LETHBRIDGE; FU, 2004) that dedicate compression of the volume of traces, making the understanding of structure easier.

Therefore, dynamic analysis has advantages that make its use beneficial. Among such advantages for example, are information accuracy on the system behavior and a goal-oriented strategy given the definition of execution scenarios, allowing for the selection of software parts of interest for analysis. As limitations of the technique, there is the problem of covering the system in the number of classes captured, due to the chosen execution scenarios. There is a difficulty in choosing which scenarios would capture all elements of interest. Another limitation

is the amount of data that affect the performance and effort of humans in dealing with the data.

From the point of view of software architecture understanding, derived representations of dynamic data have been used such for obtaining sequence diagrams (PAUW et al., 2002a), (SYSTÄ; KOSKIMIES; MÜLLER, 2001). Other approaches motivate the use of dynamic analysis for architectural recovery as shown in (HEUZEROTH et al., 2003b) and (HEUZEROTH; HOLL; LOWE, 2002) to design pattern detection and representation of relevant architecturally rules studied by (KOSKINEN; KETTUNEN; SYSTA, 2006).

In this work, we have used dynamic analysis to capture trace trees as we hypothesize that their upper level nodes are more likely to represent key classes.

2.2.2 Static Analysis

The code static analysis does not consider the inputs of a program, instead, static information is derived from artifacts that can be classes, interfaces, methods and variables and relationships that can be extension between classes or interfaces, calls between methods, etc.

Static analysis can insure a complete coverage of the program branches (CHESS; MCGRAW, 2004), used APIs, program dependencies, or the configuration files explored. Static analysis refers to different methodologies, including model checking and model provers, to verify execution paths of a program without actually executing it (PISTOIA et al., 2007). Unlike manual review, which relies on the tedious examination of sequences of the concrete or symbolic execution program, static code analyzers can capture comprehensive and accurate models of the software, like for instance an abstract representation of all the execution paths to be covered.

Struture101[1] is a tool used in our approach to obtain dependency graphs from key classes in a static context. We choose that due to increased number of features that are performed and through such being able to report a greater number of dependencies between classes compared to other available tools as shown in the study by (PRUIJT; KÖPPE; BRINKKEMPER, 2013). It is free for use on open source projects. It is used to analyze, monitor and control the software architectures. The code-base are compressed and are organized into higher-level abstractions (functions, classes, files, packages, jars, etc.), and the dependencies that emerge through this organization. It is based on diagram to define modules. The rules and violations are shown in these diagrams, with textual reports provided.

We have used static analysis to retrieve dependency graphs that were used to organize the keys classes, making explicit dependency relationships that are omitted in the documentation, in some cases.

[1] https://structure101.com/

2.3 Program Comprehension

Program comprehension is characterized by theories aimed at providing rich explanations about how programmers comprehend software, as well as tools that are used to assist in comprehension tasks.

Understanding software internal processes requires the investigation of its artifacts, such as the source code and documentation to achieve a sufficient level of knowledge. However, most programmers spend more than 50% of their time just to understand the source code (MAALEJ et al., 2014).

There are several theories that elaborate explanations regarding how programmers comprehend programs to collaborate through knowledge and experience, providing data on how the tools and methods of comprehension programs could be improved. In this sense, a lot of tools that exploit the features distinct to the programs and programmer's abilities emerged (STOREY, 2005).

In general, program comprehension tools are classified according to three main categories: extraction, analysis and presentation. Tools in the context of extraction include analyzers and instruments to collect data. The analysis tools perform static and dynamic analysis to support activities such as clustering, feature location, domain analysis, calculations, etc. Finally, presentation tools include code editors, browsers, hypertext and views. Integrated development environments and software reverse engineering, usually have some features of each said category. The supported feature set is determined by tool purpose or the research focus.

One difficulty encountered is related to how to classify such tools, i.e. how to find the main motivation of these tools, according to the different features they possess. For example, the Rigi system (MÜLLER; KLASHINSKY, 1988) supports multiple views, cross-references (cross-cutting) and queries to support understanding (bottom-up) (SHNEIDERMAN; MAYER, 1979). Bootom-up implementation refers to permit low-level code to be generated first in an attempt to in an attempt to build up to the goal. This process, referred to as "working forward" or "reformulating the givens," where the "givens" include the permissible statements of the language.

Besides the approach (bottom-up) to comprehend programs, another approach that is used is the top-down. Top-down implementation refers to comprehend of the internal semantics for a problem requiring that the highest (most general) levels be set first, followed by more detailed analysis, (from the general goal to the specifics) is one technique used by humans in problem solving. (BROOKS, 1983) based on hypothesis generation and verification (MURPHY; NOTKIN; SULLIVAN, 1995). Another tool is the Bauhaus (EISENBARTH; KOSCHKE; SIMON, 2001) which has features to support clustering (identifying components) and analysis concepts. The SHRIMP tool(STOREY, 2003) provides a meta-model for navigation support integrated that for allows frequent changes between the strategies. Finally, the CodeCrawler

tool (LANZA; DUCASSE, 2001) uses metrics visualization to support the understanding of systems and to identify gaps and other architectural features.

Strategies to improve quality of comprehension tools are described below.

Recommendation Systems. One way to improve the quality of comprehension tools is to enhance the user interface aspects, (e.g., create intelligent tools with any domain or user's knowledge). Recommendation systems are used to guide the navigation on the software. Examples that use this technology include Mylar (KERSTEN; MURPHY, 2005). Mylar uses an interest model to filter out non-relevant files in the Eclipse IDE. NavTracks provides recommendation files that are related to those who were selected by the user. Deline et al. also discuss a system to improve navigation (DELINE et al., 2005). The FEAT tool suggests using a graph (explicitly created by the programmer) to improve navigation efficiency and improve understanding (ROBILLARD; MURPHY, 2003).

Adaptive Interfaces. Another area of research includes adaptive interfaces. Software tools typically have many features that can be complex, not only for novice users, but also for experienced users. The volume of displayed information can be reduced through the use of adaptive interfaces. The idea is that user interface adapts itself to suit different types of users and tasks. Adaptive interfaces are common in Windows applications, as Word. The Eclipse IDE has several views for novice users (as Gild and Penumbra (STOREY et al., 2003)). Visual Studio has the express configuration for novice users. However, none of these conventional tools have the ability to self adapt or be easily adapted from novice user to experienced users.

Software Visualization. In the field of software visualization tools, these have been the subject of a lot of research over the past few years. Many views, most based on graphs, have been proposed to support comprehension tasks. Some examples include the research tools Seesoft (BALL; EICK, 1996), Bloom (REISS, 2001), Rigi(WONG et al., 1995), (PENNY, 1993), sv3D (MARCUS; FENG; MALETIC, 2003), and CodeCrawler (LANZA; DUCASSE, 2001).

Collaborative Support. Software teams are growing in size and becoming more distributed. In this sense, collaboration tools that support distributed software development activities are crucial. Collaborative software engineering tools have been proposed, such as Jazz and Augur (HUPFER et al., 2004) (FROEHLICH; DOURISH, 2004). There are also some tools deployed in the industry, such as CollabNet, but they are simple tools to support communication and collaboration, such as version control, email and instant message. Current tools focused industry have advanced collaboration features such as shared editors for example. Although collaborative tools for software engineering have been a research topic for several years, there has been a lack of adoption of many of these approaches, such as common editors in the industry and lack of empirical work on the benefits of these tools. The work of O'Reilly et al. (O'REILLY; BUSTARD; MORROW, 2005) proposed a command console based on a room to share views of the coordination team.

Many of the techniques of program comprehension are intended to assist developers to conduct for example, software evolution activities. However, comprehension of software applications is often hampered by the lack of documentation. Also, when there is documentation, there is no assurance that it is up to date or complete (LETHBRIDGE; SINGER; FORWARD, 2003). In these cases, the code is the main source for reliable information extraction. Several techniques have been proposed to facilitate the understanding of software systems from the source (CORNELISSEN et al., 2009). However, there is still no widely accepted approach that allows for a quick understanding of the implementation of a software feature.

In our approach program comprehension aligned with reverse engineering help to understand software behavior from the analysis of data generated during program execution. Tools for the system analysis of object-oriented execution traces were proposed (PAUW et al., 2002b), (RICHNER; DUCASSE, 2002). However, many of these tools suffer from the problem related to trace sizes, requiring significant effort from developers to visualize and understand the available data. In order to filter out irrelevant data for the proposed analysis, we can rely on classification algorithms. We use Naïve Bayes classification algorithm as an alternative to reduce the amount of trace data.

2.4 Classification Techniques - Naïve Bayes

Data mining is a process that uses algorithms to analyze in an effective way large database for extracting knowledge. One of the most useful data mining tasks is called classification.

Classification is the process of finding, via machine learning, a model that describes different data classes (HAN, 2005). The model is derived based on the analysis of training data (i.e., data objects for which the class labels are known). The model is used to predict the class label of objects for which the the class label is unknown. The purpose of the classification to label automatically new instances of the database with a particular class or function by applying a model. This model is based on the value of the attributes of the instances of training. Several classifiers have been proposed in recent years. Some use decision trees to label records. Other algorithms based on artificial neural networks use probabilistic models (Bayesian) or rules.

Naïve Bayes (HAN, 2005) is a classification technique based on Bayes' theorem with an assumption of independence among predictors. It assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an orange if it is orange color and round. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an orange.

Bayes theorem provides a way of calculating posterior probability $P(c|x)$ from $P(c)$, $P(x)$ and $P(x|c)$. Consider the equation $P(c|x)=\frac{P(c|x)P(c)}{P(x)}$. Where:

- $P(c|x)$ is the posterior probability of class (c, target) given predictor (x, attributes);
- $P(c)$ is the prior probability of class;
- $P(x|c)$ is the likelihood which is the probability of predictor given class;
- $P(x)$ is the prior probability of predictor.

A Naive Bayes algorithm works converting the data set into a frequency table. Next, creates Likelihood table by finding the probabilities. Finally, Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

In our approach, Naive Bayes computes the probability $P(c|d)$ of a trace subtree belonging to a particular class from the a priori probability $P(c)$ to be a subtree of this class and the conditional probabilities $P(t_k)|c$ of each feature t_k that occurs in a subtree of the same class. The goal of the algorithm is to find the best class C_{map} for a subtree maximizing the posteriori probability.

The classification function accepts as parameters, test subtrees, the set of classes and estimated probabilities in training. For each class a posteriori probability is calculated by adding the logarithm of the priori probability with the logarithms of the conditional probabilities of each subtree of the test set. The subtree is then labeled with the class that receives the highest posteriori probability.

2.5 Recall and Precision

In this work, we will use recall and precision measures to evaluate the effectiveness of the approach in terms of number of key classes recovered.

Precision and recall are the basic measures used in evaluating strategies such as search. In this case, there is a set of records in the database which is relevant to the search topic. So, records are assumed to be either relevant or irrelevant.

Recall is the ratio of the number of relevant records retrieved to the total number of relevant records in the database. It is usually expressed as a percentage. So, the equation $Recall = \frac{A}{A+B} * 100\%$. Where:

- A: number of relevant records retrieved;
- B: number of relevant records not retrieved;

Precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. It is usually expressed as a percentage. So, the equation $Recall = \frac{A}{A+C} * 100\%$. Where:

- A: number of relevant records retrieved;
- C: number of irrelevant records retrieved;

A measure that combines precision and recall is the harmonic mean of precision and recall, the traditional *F-measure* or balanced *F-score*: $F=2*\frac{Recall*Precision}{Recall+Precision}*100\%$.

This measure is approximately the average of the two when they are close, and is more generally the harmonic mean, which, for the case of two numbers, coincides with the square of the geometric mean divided by the arithmetic mean.

2.6 Bad Smells

Code smells is one of the concepts that limits the code quality. A code smell (MäNTYLä; LASSENIUS, 2006) dues not necessarily mean that software components contain bugs, but indicates potential weaknesses in the project that can slow down development, increasing the risk of errors or failures in the future. Common examples of bad smell code consists of code clones, very long classes and methods, very long parameter list, complex control structures, dependencies between components, etc.

The study of (BROOKS, 1995) describes how the properties of software (complexity, conformity, changeability, and invisibility) make its design an "essential" difficulty. Good design practices are fundamental requisites to address this difficulty and accordingly smells that can manifest as a result of design decisions.

Smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality (FOWLER et al., 1999). So, a designer has to analyze the smells found in a design, determine the problems underlying the smells, and then identify the required refactoring to address the problems.

Technical debt is the term used to define wrong design decisions (FOWLER et al., 1999). So, one of the indicators of technical debt is poor software quality. For example software appears complex and hard to comprehend, and has "changeability", "extensibility", "reliability" and "reusability" that is seen as detrimental. To improve software quality and reduce technical debt is discovering and addressing smells in a design software. So, there are design factors that can cause a smell to occur and thus it is necessary to take care of smells because it negatively impact software quality, and poor software quality indicates a technical debt.

Since smells may have an impact on design quality, it is important to understand smells and how they are introduced into software design. We would like to point out that since design smells contribute to technical debt, there is some overlap in the causes of design smells and technical debt.

There are distinct kinds of smells reported in the literature (FOWLER et al., 1999). Several

tools are available for detecting smells in source code (FONTANA et al., 2011). In our approach we have used DECOR (MOHA et al., 2010) as it is considered one of the state-of-art tools for detecting smells and can detect a large number of smell kinds (TUFANO et al., 2015). Next, we want to investigate if bad smells are associated with a lower cohesion and higher coupling on key classes.

2.7 Software Metrics

Programming complexity (or software complexity) is a term to describe the interactions between a number of entities. As the number of entities increases, the number of interactions between them would increase exponentially. Higher levels of complexity in software increase the risk of unintentionally interfering with interactions, and so increases the chance of introducing defects when making changes.

Software metrics are defined by measuring some property of a software portion or its specifications. Software metrics provide quantitative methods for assessing software quality, and can be used as proxies to characterize how difficult a program is to comprehend and work with (DEBBARMA et al., 2013). Software metric is a measurement, usually using numerical ratings, to quantify some characteristics or attributes of a software entity. (CHIDAMBER; KEMERER, 1994) presented a metrics suite for object oriented design. Some of the metrics are considered and described in our approach as they are more related to analysis of cohesion and coupling to measure the complexity of classes and methods.

- CBO - Coupling between object classes. The coupling between object classes (CBO) metric represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
- RFC - Response for a Class. The metric called the response for a class (RFC) measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object).
- LCOM - Lack of cohesion in methods. A class's lack of cohesion in methods (LCOM) metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields. Although, LCOM has been criticized on how it actually represents cohesion, it can be analyzed under its own definition.
- Ca - Afferent couplings. A class's afferent couplings is a measure of how many other classes use the specific class. Coupling has the same definition in context of Ca as that used for calculating CBO.

2.8 Concluding Remarks

In this chapter, we provided the background necessary to comprehend the approach that will be presented in this thesis.

In Section 2.1 we reviewed the importance of architecture recovery because we aim to use these concepts to motivate the key classes recovery in an application.

In Section 2.2 was emphasized the use of reverse engineering as a widely used solution for software architecture recovery. In the context of reverse engineering, we are going to propose an approach that combines static and dynamic analysis. We highlight the main differences between the two because we aim to use their concepts and the finality of both concerning the development of our technique.

In Section 2.3 our main goal is to recover and understand important code elements. We reported problems that limit software comprehension. In our approach, we developed several data filtering mechanisms to reduce the effort of comprehension. We highlight software visualization techniques because we aim to use these concepts to present a visual organization of key classes emphasizing dependency relationships.

In Section 2.4 we reviewed a brief explanation about Naïve Bayes concepts because it is used for classification of key subtrees.

In Section 2.5 we reviewed recall and precision because we aim to use these concepts to evaluate our approach.

In Section 2.6 and Section 2.7 we reviewed fundamentals of bad smells and software metrics because we aim to use these concepts to investigate structural problems that may exist in key classes.

In the next chapter, we describe Keecle for recovering key classes. We present the phases to extract key classes and the results achieved in 14 real-world Java systems.

Chapter 3

Keecle – Mining Key Classes Using Dynamic Analysis

The cost and effort needed to understand and adapt internal elements of software systems is related to the investigation of artifacts such as source code and documentation. Moreover, in many cases, documentation concerning design decisions is missing, or when it exists, it is neither updated nor complete. In that case, developers are required to analyze the source code, which is the only source of reliable information to understand the software architecture. Traditionally, software architectures are documented in a package based structure, since it is easier to map to actual artifacts. However, quite often this is not the best architectural organization (GARCIA et al., 2013), and when the architectural documentation is available, it is often outdated because of phenonema, such as, architectural drift or erosion (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). To alleviate these problems, several architecture reconstruction techniques have been proposed (DUCASSE; POLLET, 2009a), but a number of problems hinder the use of these techniques.

In a work that closely relates to ours, Zaidman and Demeyer (ZAIDMAN; DEMEYER, 2008) proposed a technique which can identify the most important classes in a system—the key classes. They characterized these key classes as typically having a lot of “control” within the application. In order to find these “controller classes", they presented a detection approach that is based on dynamic coupling and webmining, obtaining a precision of around 50%. In our approach, the concept of “key classes" can be mapped to those that are central for defining the meaning of an architectural design. In this chapter we present the approach to mine key classes and results in terms of recall and precision.

3.1 Outline of the Approach

Our goal is to provide an approach with higher accuracy that recovers execution trace subtrees whose roots are calls to methods for those key classes. Our hypothesis is that architectural

components can be matched to subtrees of execution traces that have a larger number of distinct method calls, which are typically near to the main tree root, i.e., they are low-depth nodes in the method call tree.

Several processes are used to implement this approach: capture traces; compress traces; discard identical subtrees; and filter the architectural relevant classes using Naïve Bayes classification algorithm.

The contributions of this chapter are twofold:

- we propose Keecle, a novel technique for the identification of architecturally relevant classes of a software system that can be provided for developers as a high-level overview to help understanding and maintenance activities;
- we provide an empirical evaluation of Keecle using open source and proprietary systems showing that it outperforms previous work, and also the generality of the approach along a consistent adequate accuracy.

3.2 The proposed approach

In general, architectural components are difficult to identify in large systems from source code with high accuracy (GARCIA; IVKOVIC; MEDVIDOVIĊ, 2013a). Then, we propose to semi-automatically identify architecturally relevant classes named as *key classes* in execution trace trees, claiming for an alternative way to understand the software architecture from a reduced set of key classes. Tahvildari and Kontogiannis (TAHVILDAR; KONTOGIANNIS, 2004) defined key classes as:

"... the classes that implement the key concepts of a system. Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties. These classes which we called key classes manage a large amount of other classes or use them in order to implement their functionality."

Their idea that *very few key classes* implement the concepts of a system motivated us to match this notion of key classes with those classes that are typically used by developers to explain a software architecture. These classes in general have strong control over the system and rely on other classes to implement software features. Our hypothesis is that key classes can be automatically identified from call trees constructed during the system execution, where the tree nodes are method calls. The key classes are expected to be near the roots (or subroots) of large execution trace subtrees that contain a large number of method calls (nodes) from distinct classes and packages, because there are dependency relationships distinct and important that denotes a strong control on the software from those roots.

Figure 2 provides an overview of the proposed approach, aided by a set of tools, organized into three phases which are presented in the following subsections.

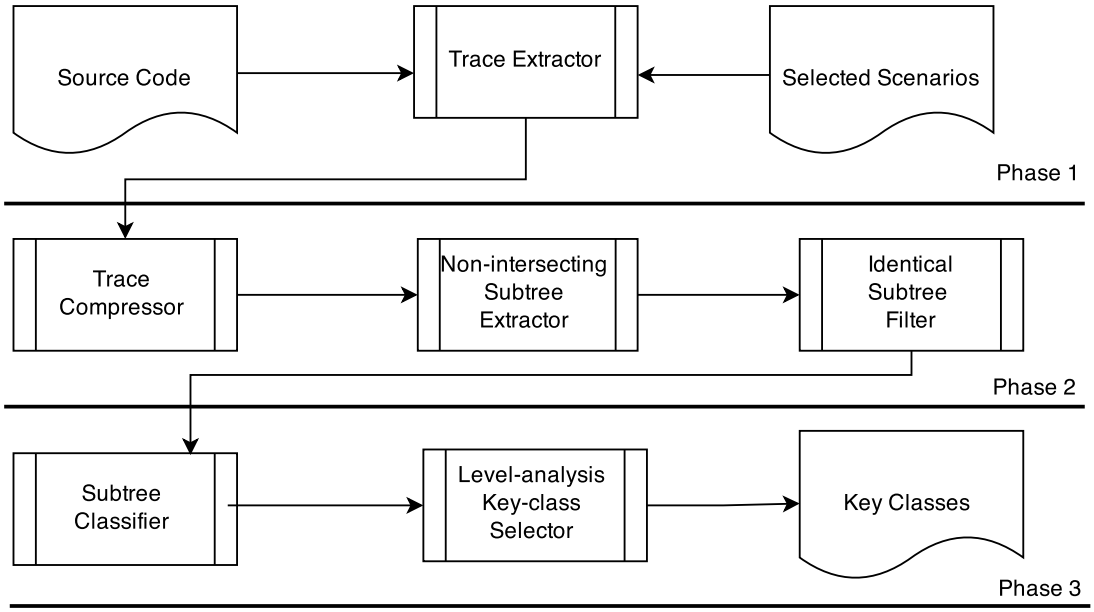


Figure 2 – Overview of the approach.

3.2.1 Phase 1 - Capturing Traces

In this work, we use the term *feature* as a functionality that can be described from the user point of view or as an observable behavior of the system that can be triggered by the user (EISENBARTH; KOSCHKE; SIMON, 2003). Whenever developers aim at comprehending software internals, we expect that they already know their main features. Our approach, as any other based on dynamic analysis, requires choosing features that are expected to cover all components of target system. In this case, our approach suggests the selection of the most representative features of the system, and it is part of the approach use to verify if those selected features are able to capture the key classes.

The target system is instrumented with *Trace Extractor* (SOBREIRA; MAIA, 2008), an AspectJ-based tool to collect the executed methods. During the execution scenario of each feature, trace files are created for each triggered thread. Each line of the trace file corresponds to a method call, which has the name of the qualified method and the corresponding level in the call stack that enables to construct a method call tree – a *Trace Tree*.

3.2.2 Phase 2 - Reducing the Size of the Traces

In this section, we present the three steps conducted in the trace reduction process. Algorithm 1 is the pseudocode for extracting reduced subtrees from execution traces.

Algorithm 1: Trace Reduction Process for a Set of Trace File *TF*

```
   Input: A set of trace files TF;
1  init
2      subTreeList ← TraceCompressor(TF)
3      halfDepth ← maxDepth(subTreeList) ÷ 2
4      NISubtreeExtractor(subTreeList,greatestSubTree(subTreeList),1,halfDepth)
5      IdenticalSubtreeFilter(subTreeList)
6      return subTreeList
7  function NISubtreeExtractor(subTreeList, greatestST, maxExpandedLevel, halfDepth)
8      if (maxExpandedLevel < halfDepth) then
9          subTreeList.remove(greatestST)
10         subTreeList.add(greatestST.children())
11         if (maxExpandedLevel < subTreeTarget.level + 1) then
12             maxExpandedLevel ← greatestST.level + 1
13         NISubtreeExtractor(subTreeList, greatestSubTree(subTreeList),
14                 maxExpandedLevel, halfDepth)
```

3.2.2.1 Compressing Traces

In the second phase, traces are compressed removing parts that are identical, typically because of loops or recursion in method calls (HAMOU-LHADJ; LETHBRIDGE; FU, 2004). So, the expected result of this compression is that the resulting larger subtrees contain more calls to distinct methods, instead of an absolute higher number of calls that could represent high number of calls to a few distinct methods.

3.2.2.2 Extracting Non-intersecting Trace Subtrees

Our rationale is that the root node of a subtree or subroots near to the root are more likely to indicate a key class that helps to understand an architecture. Moreover, nodes near the leaves of the subtree are more likely to represent fine-grained, not architecturally relevant actions, although we agree that exceptions may occur.

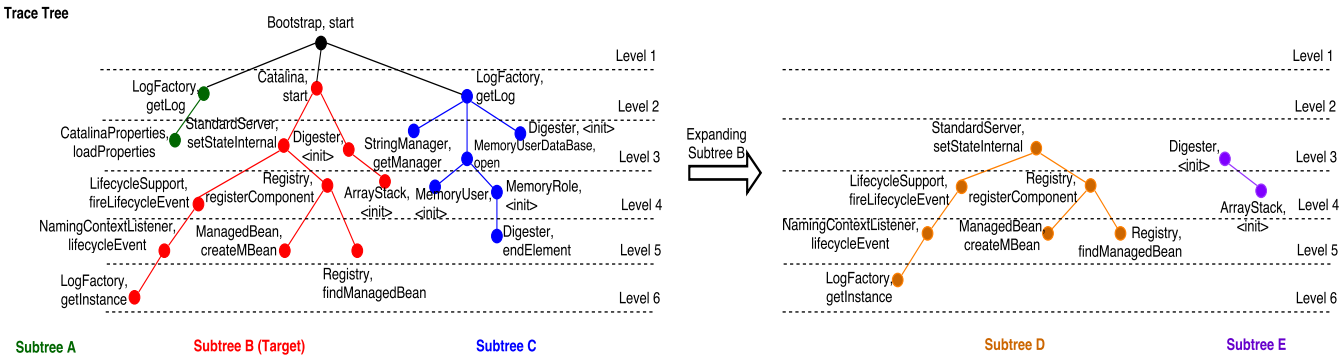


Figure 3 – Extracting code elements (subtrees) from execution traces.

The proposed method is based on the extraction of large subtrees without a non-proxy root. Proxy roots are those with only one child (method call) or important children (large subtrees)

and other non-important children (small subtrees). For each level, a set of subtrees' roots are analyzed based on their size and the size of their children.

The subtree recursive analysis process descends the trace tree, identifying and extracting subtrees. Algorithm 1 selects firstly the largest subtrees to try and split it in smaller subtrees. The recursive process ends when there is a subtree with a root in the level that corresponds to half of the original trace tree depth (50%). When half the level corresponds to a non-integer value, the value is rounded up to a higher value. Initially, we tested different stopping criteria (25%, 50%, 75% and 100%). In our preliminary results, we observed that capturing roots by selecting stopping criteria between 100% or 75%, we can have key classes referring to initialization of the system. Meanwhile, the roots extracted considering 50% stopping criteria are more specialized representing different parts (components) of the system design. However, depending on the developer's level of interest, the tool can be calibrated to extract roots that cover different aspects of the system. This choice for stop-criterion (50%) also was based on the observation that subtrees of interest: typically large and with roots or subroots near to the original tree root.

Figure 3 shows an example of this extracting process (considering stopping criteria 50%). It shows a *trace tree* where nodes are method calls. The trace tree has size 20 and contains six levels. In this case, the limit for the expansion process is at level 3, because of the defined threshold of half the maximum height.

Starting from the root *Bootstrap.start*, three child subtrees are identified with sizes: *2, 10* and *7* respectively, from the left to right on level 2. The first subtree (green subtree) has root *LogFactory.getLog*, which we named as subtree *A*. The second subtree (red subtree) has root *Catalina.start*, and was named as subtree *B*. The third subtree (blue subtree) has root *LogFactory.getLog*, and we named as subtree *C*.

The roots of the new subtrees are on level 2, then the extraction process continues on level 3. The algorithm chooses the largest subtree to split it into smaller subtrees, namely the subtree *B* with size 10. When subtree *B* is split, two new subtrees are analyzed (subtree *D* and *E*). Subtree *D* (orange subtree) has root *StandardServer.setStateInternal* with size 7, and the subtree *E* (purple subtree) has root *Digester.<init>*, with size 2. Subtree *D* and *E* have roots that are on level 3 of the trace tree, so the extraction process ends with four selected subtrees (A, C, D and E).

3.2.2.3 Filtering Identical Subtrees

Subtrees that are identical to each other are filtered out to reduce the amount of information to be analyzed. This situation occurs because algorithm that removes loops may not remove all possible loops. When the discarding process of the identical subtrees is finished, we have a limited and less complex set of subtrees that are organized in terms of features and its threads according to the execution scenario.

3.2.3 Phase 3- Classifying Trace Subtrees

The previous process is expected to reduce significantly the number of method calls for analysis. However in some cases, this number is still relatively high, due to the number of features or the complexity of the target system being high. So, we propose a classification mechanism based on a set of observable attributes of the remaining subtrees aiming at selecting the most relevant subtrees representing the key classes.

3.2.3.1 Defining the Attributes

In order to obtain an accurate classification in supervised learning, it is important to choose relevant attributes to filter desired subtrees.

In order to construct the classifier, we manually created a training and testing set, analyzing the resulting subtrees returned in the previous phase. We defined five attributes for classification of trace subtrees:

Size of subtrees: Larger subtrees are more likely to provide/consume different and important services. A relevant subtree would likely to have size that is larger than the mean size of the subtrees system's. In particular, the attribute related to size, seems to be most discriminative because the others could be somewhat dependent on it.

Distance of the subtree to the main root: subtrees near to the root of the original tree tend to represent more higher-level abstractions.

Number of distinct packages: a subtree with high package variability represents a notion or distinct relationships and are not strongly adherent to the package-based structure.

Number of distinct classes: a subtree that contains many distinct classes suggests that it encapsulates more varied responsibilities.

Number of distinct methods: the presence of distinct methods, in the same way as distinct classes in a subtree could be a sign of coarse-grain responsibility.

3.2.3.2 Classifying Subtrees

We aim at classifying subtrees into two categories: key and non-key candidates. Even if this classification process could have been applied in earlier stages of the approach, it seemed more coherent to apply it after the removal of redundant calls, to have less noise in the tree topology for the classification. This is due to the fact that we are interested in the variability of packages, classes and methods of a tree and not only in the absolute size where repetitive calls would be a noise for the classification process. For each subject system, we extract the attribute values of all subtrees. A candidate subtree should possesses roots that can become a key class. The size of this candidate must be superior or equal to the average of the size of all subtrees. For the training data, we generated distinct training groups, the data were extracted from subtrees of

the softwares in exception to the target system data in order to test the target systems, as shown in the Table 1 (leave-one-out strategy). In our experiments, the evaluation was performed using with two classifiers available on Weka software: Naïve Bayes and Neural Networks (using backpropagation). The two classifiers had very similar results in terms of correctly classified instances (99.3% and 96.5%, respectively). So, we used Naïve Bayes because it was able to select more subtrees.

Table 1 – Training and testing data.

Training Data	**Testing Data**	
Instances	**Target System**	**Instances**
473	JMeter	377
475	Ant	9
415	Lucene	69
495	Tomcat	167
633	JavaCC	11
473	Javac	29
1.219	Financial	387
1.219	Service Order	6.227
475	Scholar	257
1.219	PDFBox	839
1.219	JEdit	1.038
302	Xerces	109
302	Log4j	83
633	Jetty	486

3.2.3.3 Selecting the Key Classes based on Level-Analysis

After defining the key subtrees, the final process is to select the key classes from the key subtrees. Although, the subtree root is a good candidate for a key class, there might be other key classes in subtrees, depending on the interest of the developer in understanding the architecture with more or less details.

One question associated with the proposed technique is how to determine a target of k key classes that the approach needs to retrieve. In a real comprehension activity, developers do not know the best value of k, as in fact there is no best value, as it depends on how much detail the developer wants to comprehend. Typically, they would want to begin with less detail (less classes) and then increase the number of classes as the comprehension process evolves. So, it is reasonable that our approach can rely on an input parameter k indicating the target number of classes. We defined that the number of roots found by the algorithm has to be equal to the number of key classes (k) that the developer expects to find.

However let us suppose we have a target of k key classes to be retrieved by the approach, and the classifier has returned $k - 1$ key subtrees. The number of roots is less than k target classes, so the algorithm would descend one more level in the tree until the number of nodes is greater than or equal to the number of key classes. However, when we descend to the next level, we increase substantially the number of classes, and, of course there would be a large gap between the desired and provided level of details would be affected. As this process can have a cumulative characteristic, i.e., for each covered level of the subtrees, it increases the number

of recovered roots (roots and subroots). In these situations, when there is a difference at most two units until the number of roots found in relation to the number of key classes, our approach stops descending. In (ZAIDMAN; DEMEYER, 2008), the authors also use this kind parameter and return as a result the percentage of the ranked classes.

But, to alleviate the above mentioned problem, we proposed a strategy that ranks candidate roots. Suppose for example, that there are three candidate subtrees, and the developer wants to retrieve only one key class ($k = 1$). To determine which root of the subtree will be selected as the key class, we used a ranking algorithm to determine an order of relevance of the discriminant attributes of the trees. In this context, the *ranker* method available in Weka[1] classified attributes assigning weights to these in the order shown in Table 2.

Initially, we constructed a data set containing the values of the attributes extracted from the subtrees of the target systems. For each subtree, the value for each attribute is recovered, and respective weights are shown in Table 2. The weights of each attribute indicate the relevance order that will be considered during the subtree selection process in the Algorithm 2. These weights were automatically obtained using *GainRatioAttributeEval* an attribute evaluators in Weka, and, in sequence it was used *ranker* search method sorts attributes according to their evaluation in Weka.

We present the algorithm 2 is the pseudocode describing this process based on level analysis. The function *extractKeyClasses* evaluates the subtrees traces classified by *naiveBayesClassifier* to find candidates roots or subroots for key classes.

In the algorithm 2, the subtree with highest value has its root extracted to define a key class, and new subtrees will be extracted from this subtrees through the expansion method *NISubtree-Extractor*.

Table 2 – Relevance of attributes.

Attribute	Relevance
size of subtrees	1
number of distinct methods	0.24
number of distinct classes	0.22
number of distinct packages	0.15
distance of the subtree to the main root	0.02

3.3 Study Setting for Evaluating Keecle

In this section, we present the subject systems used to evaluate Keecle and their respective execution scenarios to extract execution traces. The ground-truth key classes considered in this evaluation used one of the following criteria:

- They were retrieved from (ZAIDMAN; DEMEYER, 2008) - Target systems: JMeter and Ant;

[1] http://www.cs.waikato.ac.nz/ml/weka/documentation.html

Algorithm 2: Process to select the key classes from the key subtrees based on level analysis.

```
   Input:
   A set of trace subtrees subtreeList;
   The target number of key classes k;
1  init
2      SBClassified ← naiveBayesClassifier(subtreeList); /* set of subtrees classified */
3      extractKeyClasses(SBClassified, k);
4      return SBClassified;
5  function extractKeyClasses(SBClassified, k)
6      depthSubtree ← 1;
7      higherWeight ← 0;
8      auxHigherWeight ← 0;
9      tempSubtree ← null;
10     while listRoots.size() < k do
11         foreach SBClassified_i in SBClassified do
12             auxHigherWeight ← 1*sizeOfSubtree + 0,24*numberOfDistinctMethods +
                 0,22*numberOfDistinctClasses + 0,15*numberOfDistinctPackages + 0,02*DistanceOfRoot;
13             if (auxHigherWeight > higherWeight) then
14                 higherWeight ← auxHigherWeigh;
15                 tempSubtree ← SBClassified;
16         listRoots.add(extractRoot(tempSubtree, depthSubtree));
17         SBClassified.add(NISubtreeExtractor(SBClassified, tempSubtree, depthSubtree, 0));
18         depthSubtree← depthSubtree+1;
19         higherWeight ← 0;
```

- They were retrieved from available documentation - Target systems: Lucene, Tomcat, Javac, JavaCC, Jetty, Xerces and Log4j;
- They were retrieved from developers - Target systems: PDFBox, Financial, Service Order and Scholar. In this situation, developers classified a initial list of classes candidate to be a key class that our approach recovered. The number of classes was guide by number of relevant features or asking to the developers. For the proprietary systems, developers did not mention missed key classes, but because we agreed to find 10 key classes, and, for to PDFBox we considered the relevant features which cover more than half of the total number of classes in the system. In sequence, using a Likert scale (from -2: *Strongly disagree* to 2: *Strongly agree*) developers specified their level of agreement on a class to be key or non-key. A class is considered key class if it is classified as *Strongly agree* or *agree* and has Weighted average ≥ 1. After the classification we asked the following question: *Is there any class missing in the set of key classes that you consider relevant in the design/architecture level?* The PDFBox developer indicated more five potential key classes such as *PDFStreamEngine, PDFont, PDFontLike, COSBase* and *PDStream*. For the proprietary systems, developers did not mention missing key classes, but because the agreed target was to find 10 key classes. After the developers specified the ground-truth, we applied Keecle again to find classes according to ground-truth, and consequently it was possible to calculate recall and precision.

For JEdit, we did not obtain ground-truth, but it was considered in our study in reason of reasonable recall and precision obtained from other systems studied. So, the approach gave us

a margin of safety that allowed us to consider JEdit in our analysis.

Our approach requires as input the target number of classes to be recovered. This number is arbitrary: the higher the number, the higher the level of detail that developers are willing to obtain. For evaluation purposes, a fair condition was to adopt the number of classes defined in the ground-truth. Zaidman and Demeyer (2008) have chosen to retrieve 15% of the classes as the ground-truth.

We considered the following open source systems in the Java programing language:

Tomcat[2] *7.0:* is a Java web server that matches the implementation of JavaServer and Javaservlet technologies with approximately 163 KLOC; The execution scenario was loading and running an application. The application is a sample application available in the distribution[3]. The execution of that application consisted in starting the server, load Tomcat localhost, deploy the application and perform a simple test of the application.

Lucene[4] *3.0.2:* is a software with a search API for document indexing with approximately 49 KLOC; For Lucene, the execution scenario consisted of indexing files and searching through use of this index. The files used were an arbitrary simple set of text files.

JavaCC[5] *(Java Compiler Compiler) 6.1:* is a tool for generating parser to use in Java applications with approximately 43 KLOC. For JavaCC, the selected execution scenario was to generate a parser for a basic arithmetric expression grammar and its syntactic tree generator.

Javac[6] *(Java programming language compiler) 1.5:* is a compiler that reads source files written in the Java programming language, and compiles them into class files. The execution scenario was the compilation of a simple HelloWorld.java.

JMeter[7] *2.0.1:* is a Java application designed to load test functional behavior and measure performance with approximately 22.234 KLOC. For JMeter, the execution scenario was the same as used in Zaidman and Demeyer (2008), that is testing a HTTP (HyperText Transfer Protocol) connection for an arbitrary site.

Ant[8] *1.6.1:* is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications with approximately 98.681 KLOC. The execution scenario was same used in Zaidman and Demeyer (2008), that is build Ant itself.

Xerces[9] *2.11.0:* a Native Interface (XNI), it is a framework for communicating a "streaming" document information set and constructing generic parser configurations. Thus it is a processor

[2] http://tomcat.apache.org/
[3] Available at https://tomcat.apache.org/tomcat-6.0-doc/appdev/sample/
[4] http://lucene.apache.org/
[5] https://javacc.org/
[6] http://openjdk.java.net
[7] http://jmeter.apache.org/
[8] http://ant.apache.org/
[9] http://xerces.apache.org/

for parsing, validating, serializing and manipulating XML with approximately 141 KLOC. The execution scenarios were the running the samples available on the the documentation.

Log4j[10] *2.3:* creates and maintains open-source software related to the logging of application behavior and released at no charge to the public with approximately 54 KLOC. The execution scenarios were the running the samples available on the the documentation. The execution scenario was to log a debug or error message in a Java application.

Jetty[11]*:* provides a Web server and javax.servlet container, plus support for HTTP/2, WebSocket, OSGi, JMX, JNDI, JAAS and many other integrations, with approximately 472 KLOC. The execution scenario was to start a server.

PDFBox[12]*:* it is an open source Java tool for working with PDF documents, with approximately 116464 KLOC. We considered 13 features obtained from the present examples in the application source code.

JEdit[13] *5.4.0:* it is a mature programmer's text editor with approximately 130 KLOC. The execution scenarios were to exercise 10 basic and usual features such as working with files (save, open and creating files), editing text and source code, etc.

For next systems described, we omitted their real names because they are proprietaries applications of a Brazilian software development company.

Financial: it is a proprietary software that control the capital movement of a company with approximately 36.702 KLOC. We considered 10 features indicated by application owner, for instance management tuition, employee control, enrollment payment management, cash flow control, etc.

Scholar: it is a proprietary software that manages educational routine on regular schools with approximately 59427 KLOc. We considered 10 features indicated by application owner, for instance issuance of the school report card, issuance of school records, disciplines control, etc.

Service Order: it is a proprietary software that provides services, bringing agility and organization to a company with approximately 558534 KLOC. We considered 10 features indicated by application owner, for instance creation general reports, service orders (open, closed or all), etc.

The choice of those systems was guided by system relevance and the architectural documentation availability and interest of the developers to collaborate. From the analysis of the documentation or selection of the developers, we obtain a set of key classes that match the architecture that will be used to evaluate the approach in terms of recall and precision. Table

[10] https://logging.apache.org/log4j
[11] http://www.eclipse.org/jetty/ 9.3.10
[12] https://pdfbox.apache.org/ 2.0.7
[13] http://www.jedit.org/

3 shows the list of recovered classes by Keecle (consider *Kc= Key class,* ✓*=Key class and* χ=non-Key class) and the set of missed key classes during the performance of Keecle.

Lucene has 12 key classes identified in the documentation, shown in Table 3. The set of key classes covers nine different packages. The names of classes *Analyzer* and *StandardAnalyzer* suggest that they should belong to the same component. Although these classes are dependent on each other, they provide distinct services for the two features of Lucene and then were maintained in separate packages.

Table 3 lists the six main key classes of Tomcat. Tomcat is a complex application with many features, suggesting the presence of many components. In the documentation is organized into main components (represented by key classes) and the subcomponents nested in the main components. This organization of Tomcat with subcomponents reinforces our decision on how to choose the target number of classes the approach would return. We could have chosen to consider only the main components (as we did) or also to include subcomponents. We decided for only these six main components, to assess the ability of the approach in detecting the most important few key classes. All key classes except *Connector* class belong to the same package. But each of these classes provide specific services for each component.

For Javac, we identified 17 key classes distributed into seven different packages. Some of these classes, such as *MemberEnter* class are representative of a secondary component relative to *Enter* class, because it consists of a phase performed by the *Enter* class.

For JavaCC, identification of the 16 key classes shown in Table 3 was also guided by analysis of the documentation. Classes are only in two distinct packages: *parser* and *jjtree*. Each class represents a distinct component, because they provide services to several other components such as tokenization, management of error messages, the construction of syntax tree and parser, etc.

For Xerces, a ground-truth of 6 key classes (interfaces) was shown in the documentation. These classes belongs only to two *xni* and *parser* packages. These classes can be viewed as a pipeline in which information flows from a scanner, then to a validator, and then to the parser.

For Log4j presents a ground-truth of 10 key classes (concrete and interfaces), as shown on the documentation. These classes belongs to four *config*, *lookup*, *core* and *layout* packages. Basically, applications using the Log4j 2 will request a Logger with a specific name from the *LogManager*. The *LogManager* will locate the appropriate *LoggerContext* and then obtain the *Logger* from it. If the *Logger* must be created it will be associated with the *LoggerConfig* that contains either: the same name as the *Logger*; the name of a parent package, or; the root *LoggerConfig*. *LoggerConfig* objects are created from Logger declarations in the configuration. The *LoggerConfig* is associated with the Appenders that actually deliver the *LogEvents*.

Jetty presents a ground-truth of 13 key classes (classes and interfaces), as shown on the documentation these classes belongs to seven *security*, *handler*, *session*, *thread*, *ssl*, *nio* and

Table 3 – List of Recovered Classes by Keecle and List of Missed Classes.

Ant		**JMeter**		**Javac**		**Lucene**	
Recovered Classes	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**
Task	✓	TestElement	✓	Gen	✓	IndexWriter	✓
IntrospectionHelper	✓	Sampler	✓	TransTypes	✓	StandardAnalyzer	✓
ProjectHelper2$ElementHandler	✓	ThreadGroup	✓	Enter	✓	SegmentInfos	χ
RuntimeConfigurable	✓	JMeter	χ	ParserFactory	χ	NIOFSDirectory	✓
DirectoryScanner	χ	PreCompiler	✓	Attr	✓	TermQuery	✓
UnknownElement	✓	TestPlan	✓	MemberEnter	✓	FreqProxTermsWriter	χ
ProjectHelper	✓	TestPlanGui	✓	JCTree$JCCompilationUnit	✓	SegmentInfos	χ
SelectSelector	χ	TestCompiler	✓	Symbol$ClassSymbol	χ	FileDocument	✓
Target	✓	JMeterThread	✓	Type$ClassType	χ	FieldInfos	χ
-	-	JMeterTreeModel	✓	Check	✓	IndexSearcher	✓
-	-	SampleResult	✓	JavaCompiler	✓	IndexReader	χ
-	-	AssertionGui	χ	Lower	✓	TermInfosReader	χ
-	-	StandardJMeterEngine	✓	Symtab	χ	-	-
-	-	JavaSampler	✓	Todo	✓	-	-
# key classes →	10		14		17		12

Missed Key Classes

Ant	**JMeter**	**Javac**	**Lucene**
Project	JMeterGuiComponent	JavacProcessingEnvironment	IndexFiles
Main	AbstractAction	TreeMaker	SearchFiles
-	-	SourceCompleter	TopDocs
-	-	Scanner	QueryParser
-	-	ClassWriter	-
-	-	Parser	-
-	-	Flow	-

JavaCC		**Tomcat**		**Jetty**		**Xerces**		**Log4j**	
Recovered Classes	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**
Main (package parser)	✓	StandardEngine	✓	SessionHandler	✓	DeferredElementNSImpl	χ	XMLConfigurationFactory	χ
Main (package jjtree)	✓	StandardService	✓	Server	✓	DOMParser	✓	Logger	✓
JJTreeParser	✓	Catalina	χ	RequestLogHandler	χ	DOMConfigurationImpl	✓	ConfigurationFactory$Factory	χ
Token	✓	StandardServer	✓	Connector	✓	XMLNSDocumentScannerImpl	✓	DefaultConfiguration	✓
JavaFileGenerator	χ	Boostrap	χ	WebAppDeployer	χ	XMLEntityScanner	✓	PatternParser	χ
JavaFile	χ	StandardHost	✓	MovedContextHandler	✓	XIncludeAwareParserConfiguration	χ	NullConfiguration	✓
OutputFile	χ	-	-	ContextHandlerCollection	✓	-	-	Logger (package core)	✓
JavaCCParser	✓	-	-	ServletHandler$CachedChain	✓	-	-	PatternLayout	χ
ParseGen	χ	-	-	HashLoginService	χ	-	-	ConfigurationFactory	χ
JavaCharStream	✓	-	-	DefaultServlet	χ	-	-	AbstractAppender	✓
LexGen	✓	-	-	XmlConfiguration	χ				
JJTreeParserTokenManager	✓	-	-	SelectChannelConnector	✓				
JJTreeParserConstants	✓	-	-	-	-				
ParseEngine	χ	-	-	χ	-	-			
JavaCodeGenerator	✓	-	-	χ	-	-			
JJTree	✓	-	-	χ	-	-			
# key classes →	16		6		13		6		10

Missed Key Classes

Javacc	**Tomcat**	**Jetty**	**Xerces**	**Log4j**
NonTerminal	StandardContext	ThreadPool	XMLComponentManager	LoggerConfig
ParseException (package parser)	Connector	HashLoginService	XMLComponent	Filter
Node	-	SslConnector	-	StrLookup
JavaCCParserTokenManager	-	SecurityHandler	-	StrSubstitutor
ParseException (package jjtree)				

PDFBox		**Financial**		**Scholar**		**Service Order**	
Recovered Classes	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**	**Recovered Classes**	**Kc**
PDFParser	✓	LancamentoContas	✓	GerarMatricula	✓	CadastroOrdemServico	✓
FontFileFinder	χ	RelBoletoPago	χ	LancarFrequencias	✓	MovMovimentacaoviewId	χ
PDDocument	✓	CadastroMovimentacaoCheque	✓	SaidaAntecipada	✓	CadastroGrupoprodutoIF	✓
PDAnnotationTextMarkup	χ	ConRecibo	✓	CadMatrizDisciplina	✓	WinOS	✓
PDPageContentStream	✓	AlterarBoleto	✓	MntDiario	✓	WSMovItensmovimentacao	χ
PDFontDescriptor	✓	ToolBarTesoura	✓	WinEscolar	✓	PGCFactory	✓
COSWriter	✓	WinTesoura	✓	CriarHorario	✓	WSEmpresa	✓
COSDocument	✓	CadastroFinFluxoCaixa	✓	VerFaltas	✓	-	-
TrueTypeFont	✓	-	-	EntradaPosHorario	✓	-	-
PDGraphicsState	χ	-	-	-	-	-	-
PDPage	✓	-	-	-	-	-	
PDFTextStripper	✓	-	-	-	-	-	
FontFormat	χ	-	-	-	-	-	
PDMetaData	✓	-	-	-	-	-	
# key classes →	14		8		9		7

Missed Key Classes

PDFBox	**Financial**	**Scholar**	**Service Order**
PDFont	TableConsultaReciboRenderer	-	CadastroParceiro
PDFontLike	-		SisParametro
COSTree	-		-
PDTree	-		-

server packages. Basically, Jetty is the plumbing between a collection of *Connector*'s that accept connections and a collection of *Handler*s that service requests from the connections and produce responses, with threads from a thread pool doing the work.

PDFBox presents a ground-truth of 14 key classes (classes and interfaces) which were classified by the developer as being a key class. The key class set belongs to distinct packages such as *text*, *cos*, *font*, *pdmodel*, *pdfparser*, *annotation*, *state* and *ttf*. A PDF file is made up of a sequence of bytes. These bytes, grouped into tokens, make up the basic objects upon which higher level objects and structures, and the package *cos* plays this role. The organization of these objects, how to they are read, and how to write them is defined in the file structure of the PDF - *pdfparser* package is accountable for this function. Within the file structure basic objects are used to create a document structure building higher level objects such as pages, bookmarks, annotations using for instance *pdmodel* package.

Financial, presents a ground-truth of 8 key classes mentioned by developers. We recovered 8 classes from 8 distinct packages: *movimentacao*, *relatorio*, *cheque*, *boleto*, *other*, *tesoura*, *caixa* and *win*. Each package represents important structures of the software and thus they can show relevant aspects of the design.

Scholar is a small software and has few packages. It presents a ground-truth of 9 key classes mentioned by the developers. We recovered 9 classes from 3 distinct packages: *cadGeral*, *win* and *escolar*. *win* package contains classes to build Gui interfaces, *cadGeral* contains classes to record data to the database and *escolar* is a package for general classes of the application.

Finally, Order Service, presents a ground-truth of 7 key classes mentioned by the developers. We recovered 7 classes from *os*, *bean*, *grupoproduto*, *dao*, *connection*. Those packages contain classes to establish connection with database, build, Guis, etc.

For Ant and JMeter, the key classes shown in the Table 3 were evaluated considered the ground-truth available by Zaidman and Demeyer (2008). Ant application contains 10 key classes from two distinct packages and JMeter contains 14 key classes from 10 distinct packages.

Noteworthy concept is related to abstract classes and interfaces. The execution traces capture methods that were effectively called, and which are connected to an object. The class that created this object should not be an abstract class or interface. In this context, if we have in the documentation an abstract class or interface as a being key class and during the capture of traces, a concrete class that extends or implements these situations is captured, so we will consider these as a key class in the our results.

3.4 Results on Recall and Precision

In this section, we present a quantitative evaluation of our approach based on the values of precision and recall. In sequence, we describe the six phases shown on Table 4, and to which extent the six phases reduce the call tree. Due to the fact that each phase filters trees, then they may worsen recall.

- **Ph1**: total number of method calls in the traces file for each thread;
- **Ph2-1**: total number of method calls in the traces file for each thread after the compression process for removing loops and recursion;
- **Ph2-2** total number of key candidate subtrees obtained;
- **Ph2-3**: removing trace subtrees for which content is identical;
- **Ph3-1**: total number of subtrees obtained by classifying processes of the traces;
- **Ph3-2**: number of classes recovered (after possible expansion).

Table 4 – Reduction of Number of Calls by Phase.

Software	Ph1	Ph2-1	Ph2-2	Ph2-3	Ph3-1	Ph3-2
JavaCC	42.548	7.535	274	29	9	16
Tomcat	12.5837	90.769	550	167	39	6
Javac	1.072.518	591	17	11	5	17
Ant	1.357.211	624.706	15	9	7	10
JMeter	192.140	73.404	2.301	377	38	14
Lucene	80.385	49.195	403	69	18	12
Xerces	239.629	15.939	154	109	44	6
Log4j	13.004	4.172	158	83	17	10
Jetty	62.067	22.200	1.073	486	41	13
JEdit	2.999.961	229.455	3.236	1.038	676	10
PDFBox	6.306.433	138.689	6.488	839	49	14
Service Order	8.858.043	2.777.662	35.861	6.227	123	7
Scholar	137.555	2.241	901	257	4	9
Financial	531.795	3.972	737	387	10	8

Table 5 (consider *P=Precision* and *R=Recall*) shows for each phase, the impact in recall reduction, which is necessary to improve the precision. We can observe at column Ph1 that the defined execution scenarios were incomplete, except for Tomcat, Scholar, Ant and Lucene, with 100% recall. Tomcat and Lucene had recall impact on the final phases. For other systems the recall did not change during the phases. Furthermore, we can observe an expressive precision improvement, specially in phase Ph3-2.

Finally, Table 6 shows the average for the attributes of each subtree for the applications. We can note that in general, the subtrees are formed by method calls from different packages and classes which reinforces the notion of that components seems to span different packages.

Table 7 presents a comparison of Keecle and the approach presented in Zaidman and Demeyer (2008). For Ant and JMeter we preferred to use the values reported in their paper. For

Table 5 – Recall (R) and Precision (P) for Phases of the Approach.

Software	Ph1		Ph2-1		Ph2-2		Ph2-3		Ph3-1		Ph3-2	
	R	P	R	P	R	P	R	P	R	P	R	P
JavaCC	69%	0.026%	69%	0.14%	69%	4%	69%	38%	69%	69%	69%	69%
Tomcat	100%	0.004%	100%	0.006%	100%	1.09%	100%	3.59%	100%	15%	67%	67%
Javac	82%	0.0015%	82%	1.18%	76%	76%	76%	76%	59%	59%	59%	59%
Ant	100%	0.0007%	100%	0.0016%	100%	67%	100%	80%	80%	80%	80%	80%
JMeter	86%	0.006%	86%	0.017%	86%	0.5%	86%	3.18%	86%	3.15%	86%	86%
Lucene	100%	0.014%	100%	0.024%	83%	2.48%	83%	14%	83%	56%	50%	50%
Xerces	67%	0.001%	67%	0.025%	67%	2.6%	67%	3.7%	67%	9%	67%	67%
Log4j	60%	0.046%	60%	0.095%	60%	3.16%	60%	3.79%	60%	3%	50%	50%
Jetty	62%	0.012%	62%	0.036%	62%	0.7%	62%	1.6%	62%	20%	62%	62%
PDFBox	71%	0.00016%	71%	0.007%	71%	0.15%	71%	1.19%	71%	20%	71%	71%
Financial	89%	0.0013%	88%	0.18%	88%	0.94%	88%	1.8%	88%	70%	88%	88%
Scholar	100%	0.006%	100%	0.4%	100%	0.99%	100%	3.5%	100%	100%	100%	100%
Service Order	71%	0.00006%	71%	0.00018%	71%	0.013%	71%	0.08%	71%	4%	71%	71%

Table 6 – Average Attributes in Each Subtree.

Software	Size	Root Level	# Packages	# Classes	# Methods
JavaCC	80.6	10.2	2.0	4.77	61.55
Tomcat	359	9.25	10.66	26.12	52.28
Javac	71.4	9.4	5.6	13.4	24
Ant	744	8.8	4.2	12.8	26.6
JMeter	117	6.10	8	15.22	29.79
Xerces	171.15	8.57	4.95	7	23.81
Log4j	35.17	9.82	4.59	10.88	22.59
Jetty	387.21	13.82	7.8	20.73	77.41
PDFBox	45.69	19.37	2.96	7.51	20.04
JEdit	79.67	17.96	3.65	9.66	29.75
Lucene	87	7.94	4.61	16.94	33.16
Financial	21.22	21.71	3.62	7.62	12.22
Scholar	33.5	4.5	3	23	24
Service Order	413	5.11	7.3	16.12	101.17

other systems, we reproduced their approach. The ranking tool of the classes was made available by the authors contacting via email, while the coupling algorithm was implemented by the author of this thesis. In sequence, with the same execution traces used to evaluate Keecle. We can observe that all F-measure values of Keecle outperformed the results in (ZAIDMAN; DEMEYER, 2008).

JavaCC: As show in Table 4 at phase Ph1 , we observe that 42.548 method calls from in a single thread were collected. *JJTree* (from *jjtree* and *parser* packages) and *JavaCCParser-TokenManager, Node* and *NonTerminal* classes were not recovered during the capture of the traces. This situation occurred because not all the actions of the features of JavaCC were adequately exercised. In sequence, 7.535 method calls were obtained in phase Ph2-1. However, the number of nodes was still high, and thus would require great effort during analysis. During the extraction of trace subtrees process (Ph2-2), we obtained 274 subtrees. The next phase Ph2-3 consists of discarding, the subtrees which are identical to each other to discard one of them. The phase Ph2-3 resulted in 29 subtrees. During the phase Ph3-1, 9 trace subtrees were classified as shown in Table 8. The size of these 9 subtrees ranged from 79 to 100, and the call level of the first root of each of the subtrees ranged from 5 to 19. Table 6 shows the average of the attributes for each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 16

Table 7 – Recall and Precision for the Phase Ph3-2 and (ZAIDMAN; DEMEYER, 2008)'s approach.

Software	#Classes Doc	# Classes Recovered		Recall (%)		Precision(%)		F-Measure(%)	
		Keecle	(ZAIDMAN; DEMEYER, 2008)	Keecle	(ZAIDMAN; DEMEYER, 2008)	Keecle	(ZAIDMAN; DEMEYER, 2008)	Keecle	(ZAIDMAN; DEMEYER, 2008)
Javacc	16	14	9	69%	25%	69%	44%	69%	32%
Tomcat	6	14	59	67%	83%	67%	8%	67%	15%
Javac	17	17	8	59%	6%	59%	13%	59%	8%
Ant	10	12	19	80%	90%	80%	47%	80%	62%
JMeter	14	14	28	86%	93%	86%	46%	86%	62%
Lucene	12	12	19	50%	16%	50%	11%	50%	13%
Jetty	13	13	20	62%	15%	62%	10%	62%	20%
Log4j	10	10	13	50%	10%	50%	8%	50%	9%
Xerces	6	6	5	67%	50%	67%	60%	67%	54%
PDFBox	14	14	21	71%	64%	71%	43%	71%	50%
Financial	8	8	29	88%	50%	88%	13.8%	88%	21%
Scholar	9	9	12	100%	22%	100%	9%	100%	12.8%
Service Order	7	7	108	71%	43%	71%	2.8%	71%	5.25%
Mean				70.7	43.6	70.7	30.5	70.7	28

key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 69% and 69%, the value for the F-measure was equal to 69% as shown in Table 7.

Tomcat: As show in Table 8 on the phase Ph1, we observe that 125.837 method calls stored in 4 threads were collected. During the phase Ph2-1 90.769 method calls were recorded. However, the number of nodes was still high, requiring high levels of effort to analysis. During the extracting of trace subtrees process (Ph2-2), we obtained 550 subtrees. The next phase Ph2-3 consists of discarding, the subtrees which are identical to each other to discard one of them resulting in 167 subtrees. Phase Ph2-3 resulted in 39 subtrees. During the PH3-1 *StandardContext* and *Connector* were not classified. Table 3 shows the recovered roots. Thus, the recall and precision values were respectively 67% and 67%, the value of F-measure was equal to 67% as shown in Table 7.

Javac: As shown on Table 4, we observe that phase Ph1 returns 1.072.518 method calls in a single thread. *Parser* and *SourceCompleter* and *Scanner* classes were not recovered with the selected execution scenario. Phase Ph2-1 as shown on Table 4, 591 method calls were recorded. After phase (Ph2-2), 17 subtrees were obtained. The next phase Ph2-3 resulted in 11 subtrees. The *TreeMaker* class (Table 3) was discarded during the extraction process in the phase Ph2-2. During classifying trace subtrees, 5 trace subtrees were classified as shown on Table 4 - phase Ph3-1. The size of those 5 subtrees ranged from 9 to 71, and the call level of the first root of each of the subtrees ranged from 7 to 11. Table 6 shows the average number of attributes for each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 17 key classes applied to Algorithm 2. *JavacProcessingEnvironment* class was not classified in the phase Ph3-1. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 59% and 59%, the value for the F-measure was equal to 59% as shown

in Table 7. *Flow, Scanner* and *ClassWriter* classes were not roots selected during the final phase.

Ant: As show in Table 4 - phase Ph1 collected 1.357.211 method calls in 4 threads were collected. Phase Ph2-1 as shown on Table 4, recovered 624.706 method calls were recorded. However, the number of nodes was still high, requiring high levels of effort when analyzing. During the extracting of trace subtrees process (Ph2-2), we obtained 15 subtrees. The next phase Ph2-3 that discards, identical subtrees, resulted in 9 subtrees. During the process of classifying trace subtrees in phase Ph3-1 7 trace subtrees were selected as shown in Table 4. The size of these 7 subtrees ranged from 41 to 2.011, and the call level of the first root of each of the subtrees ranged from 5 to 15. Table 6 shows the average for the attributes of each subtree. *Main* and *Project* classes were not retrieved by the classifier. In order to perform phase Ph3-2, our input parameter corresponds to 10 key classes applied on Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 80% and 80%, the value of the F-measure was equal to 80% as shown in Table 7.

JMeter: As show on Table 4 at phase Ph1, we observe that 192.140 method calls stored in 10 threads were collected. *JMeterGuiComponent* and *AbstractAction* classes were not recovered during the capture of the traces because the feature of JMeter were not adequately exercised. Phase Ph2-1 as shown on Table 4 collected 73.404 method calls were recorded. However, the number of nodes was still high, requiring high levels of effort when analyzing. During the extracting of trace subtrees process (Ph2-2), we obtained 2301 subtrees. The phase Ph2-3 that discards, identical subtrees, resulted in 377 subtrees. During the process of classifying trace subtrees, 38 trace subtrees were selected as shown on Table 4 - phase Ph3-1. The size of these 38 subtrees ranged from 32 to 433, and the call level of the first root of each of the subtrees ranged from 2 to 12. Table 6 shows the average for the attributes of each subtree. In order to perform Ph3-2 phase, our input parameter corresponds to 14 key classes applied on Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 86% and 86%, the value of the F-measure was equal to 86% as shown in Table 7.

Lucene: As show on Table 4 - phase Ph1, we observe that 80.385 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 49.195 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 403 subtrees. The *IndexFiles* and *SearchFiles* classes were roots discarded during the extracting process (Table 3). The phase Ph2-3 that discards, identical subtrees, resulted in 69 subtrees. In the phase Ph3-1, 18 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 46 to 252, and the call level of the first root of each of the subtrees ranged from 3 to 13. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 12 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 50% and 50%, the value of the F-measure was equal to 50% as shown on Table 7. *TopDocs* class is a leaf

node and *QueryParser* class was not a root selected.

Xerces: As show on Table 4 - phase Ph1, we observe that 239.629 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 15.939 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 154 subtrees. The phase Ph2-3 that discards, identical subtrees, resulted in 109 subtrees. In the phase Ph3-1, 44 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 32 to 5.135, and the call level of the first root of each of the subtrees ranged from 3 to 9. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 6 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 67% and 67%, the value of the F-measure was equal to 67% as shown on Table 7. Two interfaces were not captured during execution traces: *XMLComponent* and *XMLComponentManager*. *DOMParser* class extends *AbstractDOMParser* class. This implements *XMLDocumentHandler, XMLDTDHandler* and *XMLDTDContentModelHandler*. Finally, *DOMConfigurationImpl* class implements *XMLParserConfiguration*.

Log4j: As show on Table 4 - phase Ph1, we observe that 13.004 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 4.172 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 158 subtrees. The *Filter*, *StrLookup* interfaces and *StrSubstitutor* class did not have captured among the traced concrete classes (Table 3). The phase Ph2-3 that discards, identical subtrees, resulted in 83 subtrees. In the phase Ph3-1, 17 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 6 to 165, and the call level of the first root of each of the subtrees ranged from 2 to 16. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 10 key classes applied to Algorithm 2, in this phase *LoggerConfig* class was not classified. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 50% and 50%, the value of the F-measure was equal to 50% as shown on Table 7.

Jetty: As show on Table 4 - phase Ph1, we observe that 62.067 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 22.200 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 1.073 subtrees. The *SslConnector* interface did not have captured among the traced concrete classes (Table 3). The phase Ph2-3 that discards, identical subtrees, resulted in 486 subtrees. In the phase Ph3-1, 41 trace subtrees were classified as shown on Table 4 in this phase was missed *QueuedThreadPool* a concrete class that implements *ThreadPool* and *HashLoginService* class. The size of the subtrees ranged from 1 to 2.006, and the call level of the first root of each of the subtrees ranged from 3 to 26. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 12 key classes applied to Algorithm 2 in this phase *SecurityHandler* class was not selected. Table 3 shows the recovered and missed roots. Thus,

the recall and precision values were respectively 62% and 62%, the value of the F-measure was equal to 62% as shown on Table 7.

JEdit: As show on Table 4 - phase Ph1, we observe that 2.999.961 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 229.455 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 3.236 subtrees. The *XMLComponentManager* and *XMLParserConfiguration* interfaces did not have captured among the traced concrete classes (Table 3). The phase Ph2-3 that discards, identical subtrees, resulted in 1.038 subtrees. In the phase Ph3-1, 676 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 2 to 235, and the call level of the first root of each of the subtrees ranged from 1 to 49. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 10 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots.

PDFBox: As show on Table 4 - phase Ph1, we observe that 6.306.433 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 138.689 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 6.488 subtrees. The *PDFont* abstract class and *PDFontLike* interface did not have captured among the traced concrete classes (Table 3). The phase Ph2-3 that discards, identical subtrees, resulted in 839 subtrees. In the phase Ph3-1, 49 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 1 to 2.006, and the call level of the first root of each of the subtrees ranged from 3 to 26. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 14 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 71% and 71%, the value of the F-measure was equal to 71% as shown on Table 7. Considering abstract classes and interfaces captured by concrete classes we have: *PDFTextStripper* extends *PDFStremaEngine* while *PDMetaData* class extends *PDstream* and finally *COSDocument* class extends *COSBase*.

Financial: As show on Table 4 - phase Ph1, we observe that 531.795 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 3.972 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 737 subtrees. The phase Ph2-3 that discards, identical subtrees, resulted in 387 subtrees. In the phase Ph3-1, 10 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 17 to 71, and the call level of the first root of each of the subtrees ranged from 2 to 7. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 8 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 89% and 89%, the value of the F-measure was equal to 89% as shown on Table 7. In our results *TableConsultaReciboRenderer* class was not recovered this is due to the algorithm that ranks the relevant classes, during the selection of key classes.

Scholar: As show on Table 4 - phase Ph1, we observe that 137.555 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 2.241 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 901 subtrees. The phase Ph2-3 that discards, identical subtrees, resulted in 257 subtrees. In the phase Ph3-1, 4 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 27 to 40, and the call level of the first root of each of the subtrees ranged from 4 to 5. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 9 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 100% and 100%, the value of the F-measure was equal to 100% as shown on Table 7.

Service Order: As show on Table 4 - phase Ph1, we observe that 8.858.043 method calls from in a single thread were collected. Phase Ph2-1 shown on 4, 2.777.662 method calls were recorded. During the extracting of trace subtrees process (Ph2-2), we obtained 35.861 subtrees. The phase Ph2-3 that discards, identical subtrees, resulted in 6.227 subtrees. In the phase Ph3-1, 123 trace subtrees were classified as shown on Table 4. The size of the subtrees ranged from 15 to 5824, and the call level of the first root of each of the subtrees ranged from 1 to 17. Table 6 shows the average for the attributes of each subtree. In order to performer the phase Ph3-2, our input parameter corresponds to 7 key classes applied to Algorithm 2. Table 3 shows the recovered and missed roots. Thus, the recall and precision values were respectively 89% and 89%, the value of the F-measure was equal to 89% as shown on Table 7. In our results *CadastroParceiro* and *SisParametro* classes was not recovered this is due to the algorithm that ranks the relevant classes, during the selection of key classes.

3.4.1 Summary of Results

Table 3 shows the list of recovered classes by Keecle (consider *kc=key class*) and Table 7 shows the results for Keecle and Zaidman and Demeyer (2008), indicating the values of precision, recall and the F-measure obtained for the systems.

An important concept to be mention concerns abstract classes and interfaces. The execution traces capture methods that were effectively called, and which are connected to an object. The class that created this object should not be an abstract class or interface. In this context, if we have in the documentation an abstract class or interface as a key class and during the capture of traces, a concrete class that extends or implements these situations was shown, so we will consider abstract classes and interfaces in the our results. For instance, on Ant, the execution traces captured *Task*, a concrete class that implements the *TaskContainer*.

In Lucene, *NIOFSDirectory* is a concrete class that extends *FSDirectory* an abstract class. *FSDirectory* class extends *Directory*. *TermQuery* is a concrete class that extends *Query* an abstract class.*FileDocument* class implements *Document*. *StandardAnalyzer* is a concrect class that implements *Analyzer*. So, we considered in our results the *Document*, *FSDirectory, Direc-*

tory, Analyzer and *Query* classes.

For JMeter, *JMeterEngine* is a interface. Keecle captured *StandardJMeterEngine*, a concrete class that implements the *JMeterEngine*. The same occured for *TestPlan* class that implements *TestElement* interface and *JavaSampler* class that implements *TestListener* So, the *JMeterEngine*, *TestElement* and *TestListener* were considered in our results.

The same situation is observed Xerces, Jetty and PDFBox. Xerces had the worst recall and precision due to the small amount of key classes required and the best result is to Scholar.

3.5 Discussion

A fundamental characteristic of this approach is the goal of retrieving classes representing an understanding architectural of a target system from subtrees of method calls extracted from execution traces. The trace compression process, the trace subtree extraction and the elimination of identical subtrees played a fundamental role because the volume of analyzable data could be adequately reduced. This fact is particularly observed in relation to the number of method calls in the experiment with Ant that was 1.357.211, after the compression process it was reduced to 624.706 and during the identical subtree removal process, only 9 subtrees remained. A similar situation occurred with the number of subtrees of JMeter, which after identical subtrees removal only 377 remained, compared to more than the previous two thousand.

An observed limitation is related to the loss of roots when the subtrees extraction process is performed. This step was responsible, for example, for the low recall in the study with Javac. In particular, the roots of interest of Javac (*JavacProcessingEnvironment*) located at level 9 and Lucene (*SearchFiles* and *IndexFiles*) classes located on level 1 of the trace tree, but due to subtrees extraction process, these roots were eliminated. Another limitation is related to the definition of adequate execution scenarios. Javac and JMeter for example, there was a reduction of recall during the trace extraction process. The choice of the scenarios did not provide good coverage of system classes, because the scenarios were simple.

The classification process on the other hand, was responsible for eliminating a significant number of subtrees. In all experiments, for example, a large number of subtrees with granularity equal to 1, 2 or 3. In this situation, the classifier was fundamental to eliminate those subtrees.

For the all considered systems in our experiment, we obtained an average for the values of F-measure of 70.7%. In our experiments, we tend to observe balanced recall and precision because our approach recovers a predefined number of key classes. Target systems contains thousands of classes and our approach was effective in reducing the number of these classes. Some systems such as Lucene, Jetty, Log4J, presented below-average recall and precision. One possible explanation, would be the simple exercise execution scenarios were not enough to cover all the ground-truth available in the documentation. As a future work, would be consider

new features during the collect of execution traces.

The F-measure average of our approach when compared with the average accuracy of the techniques presented in the work of (GARCIA; IVKOVIC; MEDVIDOVIĊ, 2013b), which corresponds to 45% (MoJoFM metric (WEN; TZERPOS, 2004)), we see a slight improvement in the final results, although it is not possible to directly compare the results due to different used metrics. In the approach presented by Zaidman and Demeyer (ZAIDMAN; DEMEYER, 2008), the case studies achieved F-measure (Mean) of 28%. Our approach achieved with those case studies F-measure (Mean) of 70.7%. One possible explanation is that the approach retrieves an exact number of key classes indicated by the user without affecting precision.

Finally, the information available in the documentation that allowed the listing of classes of interest may not be in fact the main classes of architecture, since the system presents different versions of code that are not necessarily directly reflected in the documented architecture. Moreover, the list of those classes may not be definitive. Maybe, it would be acceptable for developers to include other classes in that list. This situation is noted for example on PDFBox software, because only one developer agreed to collaborate to classify the classes set as a key or non-key class. In this case the ground-truth has a debatable degree of confidence.

3.5.1 Threats to Validity

Even with the careful planning and formal procedures applied during the execution of the experiments, some threats should be considered in the evaluation of the results validity.

External Validity: The representativeness of target systems. Although the 14 systems used in the approach where some of them are well-known systems and used in other studies, factors as the number, domain limit the generalization of our results. Other systems would generate different obstacles to the use of the approach. Other threats to external validity refers to the Java programming language, which was the only one considered in this study

Internal Validity: Although we have adopted a direct procedure to select the key classes from the documentation, different interpretations could occur on the intention of the developer to assume as class as a key one.

Construct Validity: We have used an internally constructed tool suite to run the approach. Although, the tool has been tested and verified, there still may remain some undetected bug as occurs with any software. There is no widely recognized and adopted tool support for this kind of approach, so any other adopted solution would incur a similar threat. We tried to minimize this threat checking the results in each phase.

3.6 Concluding Remarks

In this thesis proposal, we have proposed an approach based on mining of method calls to capture the notion of architecturally relevant classes. We evaluated Keecle with 14 open source systems to retrieve a reduced number of architecturally relevant classes which enables an initial understanding of the software architecture. Several phases were proposed to improve precision and recall.

One of our goals was to show that we can deal effectively with the volume of traces data, using compression techniques and removing irrelevant data. The evaluation, showed that the approach produced encouraging values of recall and precision outperforming previous work in the literature.

Next chapter, we are going to evaluate structural and ownership properties on key classes.

CHAPTER 4

Understanding Structural and Social Properties of Key Classes

In the previous chapter, we have proposed Keecle, a dynamic analysis approach for detecting key classes in a semi-automatic manner. In this chapter, we investigate some properties of key classes. Several architectural descriptions of real systems are documented using key classes. However, software documentation may have simplified descriptions from source code, without a diagnosis of the structural problems that those classes may have.

Under this motivation, we investigate if key classes are more prone to bad smells than non-key classes and if structural metrics of key classes can be associated to the occurrence of bad smells. Next, we study whether organizing keys classes in a dependency graph structure can reveal high level dependency relationships and to produce a degree of adherence with the available documentation. Finally, we analyze the ownership property of key classes.

4.1 Outline of the Study

Assessing design with all classes of the systems as a starting point is a difficult task. So, architecture reconstruction approaches were proposed to retrieve architectural components to facilitate design assessment. However, these approaches are still difficult to apply and have low accuracy (GARCIA; IVKOVIC; MEDVIDOVIĊ, 2013b). We have observed that several real-world systems such as such as Lucene[1], Tomcat[2] and Javac[3] use some few classes to document its architectural design. So, instead of recovering architectural components, we have described Keecle in the previous chapter, as a semi-automatic way for finding *key classes* considered as important design classes in object-oriented systems.

Key classes are presumably those classes that implement concepts that the developer under-

[1] https://lucene.apache.org/core/4_4_0/core/overview-summary.html
[2] https://tomcat.apache.org/tomcat-5.5-doc/architecture/overview.html
[3] http://openjdk.java.net/groups/compiler/doc/compilation-overview/

stands as the most important ones to explain the system design. The automatic finding of key classes was initially proposed by Zaidman and Demeyer (ZAIDMAN; DEMEYER, 2008), but there is still no concrete evidence that the awareness of them is a useful information for developers. So, we in investigate the role of key classes as a starting point for understanding and assessing software design.

To provide more evidence that key classes, especially those recovered by Keecle, can be a useful source of information for understanding and assessing software design, in this chapter we investigate structural and social properties of key classes:

- The first studied property is the likelihood of key classes association to bad smells. First, we analyze the proneness of occurrence of bad smells in key classes compared to the rest of the classes. Also, we analyze if the occurrence of specific bad smells are associated with different levels in cohesion and coupling metrics.
- The second studied property is related to the occurrence on circular dependency of key classes obtained from dependency relationships. We organize key classes in a dependency graph to explicitly complement and visualize circular dependencies due to the fact these they could affect the structure of the project. In addition we aim at evaluate whether dependency graphs are adherent solutions in relation to available documentation. because typically, these dependencies are neither documented nor complete.
- The third studied property is related to the ownership of key classes and thus has a social context. We evaluate the distribution of key classes among developers to understand how ownership compares to non-key classes.

The results of our study indicate that:

- Key classes manifest more often the presence of *Complex Class* code smell with respect to non-key classes of a target system. This suggests that among those classes with design anomaly symptoms, the design-relevant classes would be more likely to impact design anomaly as a whole.
- Developers would benefit from additional information about complex dependency relationship, such as circular dependencies in key classes as being a warning to maintenance activities in the future.
- Developers could prioritize code reviews of commits from ownership of the key classes to improve the overall design of the system.

4.2 Code Smell and Metrics Assessment

Bad code smells (shortly “code smells” or “smells”) are related to poor implementation and poor design choices, possibly hindering the software maintenance (BROWN et al., 1998). There

are several tools available to detect code smells and whenever possible perform refactoring operations.

Because key classes are presumably related to design, we investigate if there is any association of key classes to higher occurrence of bad smells. Moreover, to understand it key classes are critical for the design quality, we investigate if classical indicators for assessing modularity (coupling and cohesion) have different levels in key classes compared to non-key classes. We also investigate the interplay between these indicators and the occurrence of smells. For that, we pose the following questions.
RQ_1: *Are key classes more prone to the occurrence of specific bad smells compared to non-key classes?* This question aims at investigating whether key classes are more prone to bad smells and which kind of bad smells are more common in key classes.

All Java classes of the subject systems were submitted to DECOR for finding code smells. We use DECOR (MOHA et al., 2010) because as it is considered a state-of-art tool for detecting smells (TUFANO et al., 2015). The answer to that question is based on the analysis of the the relative frequency of the several kinds of smell in key classes *(kc)* compared to non-key classes *(nkc)* of the system and the gold set classes *(gs)* extracted from documentation or from developers and compared to non-gold set classes *(ngs)*, which is shown in Table 8. We consider these different classes of groups for subsequent analysis of the results between the groups. So, we can note similar results between *kc* and *gs*. Therefore, in a real situation in which the documentation is not available or outdated, keecle provides significant results to the developer to document the design.

The *ComplexClass, LongMethod* and *LongParameterList* smell kinds are the most frequent. Complex classes are more prevalent in key classes across for most systems. But, key classes are not going to naturally direct to the ComplexClass bad smell, because not all systems evaluated show this specific smell. Finally, key classes may exist in higher percentage, but not necessarily be the most complex. The *Long Parameter List* smell is not very prevalent considering the universe of methods. Moreover, differently from the Complex Class smell, no sharp differences were observed in key classes compared to non-key classes. The *Long Method* smell, similar to the *Long Parameter List* is not very prevalent considering the universe of methods, possibly indicating that although key classes seems to be more complex, their methods do not suffer much from being long.

On the other hand, there are key classes with bad smells that may have an impact that will affect some future bad-smells, such as key classes with *RefusedParentBequest, SpeculativeGenerality, SpaghettiCode* ect., and therefore should be resolved in the future during software maintenance.

Table 8 – Occurrence of smell in key classes (kc) and gold set (gs)

Bad Smell	#Occurrence		%Occurrence		# other classes		% other classes	
	gs	kc	gs	kc	ngs	nkc	ngs	nkc
JMeter								
ComplexClass	5/14	7/17	0.357	0.412	88/769	86/769	0.114	0.111
LongParameterList	2/246	5/271	0.008	0.0185	119/5520	116/5249	0.021	0.022
LongMethod	4/246	4/271	0.016	0.0148	165/5520	165/5249	0.029	0.031
SpaghettiCode	1/17	1/17	0.059	0.059	5/769	5/769	0.006	0.006
AntiSingleton	1/17	1/17	0.059	0.059	35/769	35/769	0.045	0.045
Blob	0	0	0	0	4/769	4/769	0.005	0.005
LazyClass	0	0	0	0	14/769	14/769	0.018	0.018
ClassDataShouldBePrivate	0	0	0	0	11/769	11/769	0.014	0.014
Lucene								
ComplexClass	5/12	8/12	0.417	0.471	210/2151	205/2151	0.098	0.091
LongMethod	3/702	8/501	0.004	0.016	281/10847	276/11048	0.026	0.025
LongParameterList	3/702	6/501	0.004	0.012	116/10847	113/11048	0.010	0.010
ClassDataShouldBePrivate	3/12	2/12	0.25	0.118	74/2151	75/2151	0.007	0.033
AntiSingleton	1/12	2/12	0.083	0.118	43/2151	40/2151	0.020	0.018
RefusedParentBequest	1/12	1/12	0.083	0.059	0	1/2151	0	0.000
SpaghettiCode	1/12	1/12	0.083	0.059	0	3/2151	0	0.001
BaseClassShouldBeAbstract	1/12	0	0.083	0	2/2151	3/2151	0	0.001
LazyClass	0	0	0	0	13/2151	13/2151	0.006	0.006
SpeculativeGenerality	0	0	0	0	3/2151	3/2151	0.001	0.001
ManyFieldAttrsButNotComplex	0	0	0	0	1/2151	1/2151	0.0004	0.0004
Ant								
LongMethod	7/334	6/261	0.021	0.022	145/8703	146/8876	0.016	0.016
LongParamaterList	5/334	3/261	0.015	0.011	19/8703	21/8876	0.002	0.002
ComplexClass	4/10	3/10	0.4	0.3	88/1195	89/1193	0.073	0.048
RefusedParentBequest	1/10	1/10	0.1	0.1	0	4/1193	0.000	0.002
AntiSingleton	0	0	0	0	0/1195	3/1193	0	0.001
SpeculativeGenerality	0	0	0	0	0/1195	1/1193	0	0.000
ClassDataShouldBePrivate	0	0	0	0	0/1195	9/1193	0	0.005
LazyClass	0	0	0	0	0/1195	41/1193	0	0.022
BaseClassShouldBeAbstract	0	0	0	0	0/1195	4/1193	0	0.002
Javac								
LongParamaterList	8/895	11/1321	0.009	0.008	96/7907	93/7481	0.012	0.012
ComplexClass	13/17	11/17	0.765	0.647	82/999	84/999	0.082	0.084
ClassDataShouldBePrivate	3/17	5/17	0.174	0.294	38/999	35/999	0.038	0.035
LongMethod	5/895	3/1321	0.006	0.002	79/7907	81/7481	0.011	0.011
AntiSingleton	0/17	1/17	0	0.059	16/999	15/999	0.016	0.015
BaseClassShouldBeAbstract	1/17	1/17	0.059	0.059	0	10/999	0	0.010
RefusedParentBequest	0	0	0	0	0	4/999	0	0.004
SpaghettiCode	0	0	0	0	0	4/999	0	0.004
LazyClass	0	0	0	0	0	22/999	0	0.022
ManyFieldAttrsButNotComplex	0	0	0	0	0	2/999	0	0.002
Scholar								
LongMethod	4/612	4/612	0.006	0.006	6/4.401	6/4.401	0.001	0.001
LongParameterList	1/612	1/612	0.001	0.001	5/4.401	5/4.401	0.001	0.001
LazyClass	1/9	1/9	0.1	0.1	2/415	2/415	0.004	0.004
SpaghettiCode	1/9	1/9	0.1	0.1	6/415	6/415	0.014	0.014
AntiSingleton	1/9	1/9	0.1	0.1	0	0	0	0
ClassDataShouldBePrivate	1/9	1/9	0.1	0.1	2/415	2/415	0.004	0.004
ComplexClass	2/9	2/9	0.2	0.2	4/415	4/415	0.009	0.009
Financial								
LongMethod	3/8	4/8	0.375	0.5	25/2011	24/2011	0.012	0.012
LongParameterList	3/8	3/8	0.375	0.375	12/2011	12/2011	0.005	0.005
LazyClass	1/8	1/8	0.125	0.125	13/122	13/122	0.107	0.107
ClassDataShouldBePrivate	1/8	1/8	0.125	0.125	2/122	2/122	0.016	0.016

Continued on next page

Table 8 – *Continued from previous page*

Bad Smell	#Occurrence		%Occurrence		# other classes		% other classes	
	gs	kc	gs	kc	ngs	nkc	ngs	nkc
ComplexClass	2/8	3/8	0.25	0.375	20/122	19/122	0.16	0.15
Service Order								
LongMethod	3/477	3/477	0.006	0.006	435/49.308	435/49.308	0.009	0.009
LongParameterList	3/477	4/477	0.006	0.006	221/49.308	220/49.308	0.004	0.004
AntiSingleton	1/7	1/7	0.14	0.14	64/3.354	64/3.354	0.019	0.019
ClassDataShouldBePrivate	1/7	1/7	0.14	0.14	64/3.354	64/3.354	0.019	0.019
ComplexClass	3/7	3/7	0.42	0.42	242/3.354	242/3.354	0.072	0.072
PDFBox								
LongMethod	3/14	3/14	0.21	0.21	248/8.451	248/8.451	0.029	0.029
LongParameterList	2/14	2/14	0.14	0.14	80/8.451	80/8.451	0.009	0.009
ComplexClass	9/14	10/14	0.64	0.71	203/1.160	203/1.160	0.174	0.714
Xerces								
RefusedParentBequest	0	1/6	0	0.17	90/887	90/887	0.101	0.101
LongMethod	0	3/96	0	0.03	57/8.455	60/8.156	0.006	0.007
Log4j								
LongParameterList	3/218	2/450	0.013	0.004	65/9.746	66/9.514	0.007	0.007
LongMethod	0	1/450	0	0.002	86/9.746	85/9.514	0.009	0.009
SpeculativeGenerality	0	1/7	0	0.14	1/1.472	1/1.472	0.0007	0.0007
Jetty								
LongMethod	1/137	1/115	0.007	0.009	116/12.230	116/12.252	0.009	0.009
LongParameterList	1/137	0	0.007	0	84/12.230	85/12.252	0.007	0.007
JEdit								
MessageChains	-	6/10	-	0.6	-	119/1.367	-	0.087
LongMethod	-	1/678	-	0.001	-	53/6.584	-	0.008

RQ_2: *Are key classes different in terms of cohesion and coupling metrics compared to non-key classes?* This question aims at investigating usual indicators concerning the quality of software projects, namely cohesion and coupling. We evaluated four cohesion and coupling metrics comparing those metrics within two different groups: key and non-key classes. The COPE (*Component Adaptation Environment*)(KAKARONTZAS et al., 2013) tool was used to extract the metrics *Ca* (Afferent couplings), *LCOM* (Lack of cohesion in methods), *RFC* (Response for a Class) and *CBO* (Coupling between object classes).

Figure 4 shows the distribution of the values of the cohesion and coupling metrics. We conducted the Mann–Whitney–Wilcoxon test (OJA, 2011) on all metrics and the results confirm significant differences between key classes and non-key classes ($p < 0.05$). However, there is an interesting point to observe which is although the medians are significantly different, we can observe that those metrics are not able to precisely define which classes should be considered key classes because there is a significant number of non-key classes (generally the 25% upper values) that are mostly coincident with the values for key classes. In other words, we observe that key classes are in general more prone to worse metrics, however the inverse is not in general true, i.e., a reasonable part of non-key classes (outliers) are also prone to worse metrics. On Table 9 shows descriptive statistics and p-values of tests applied on key classes and the gold set (documentation) to highlight the similarity of data for our purposes, i.e., using Keecle key classes is as good as using gold set key classes, despite of minor differences.

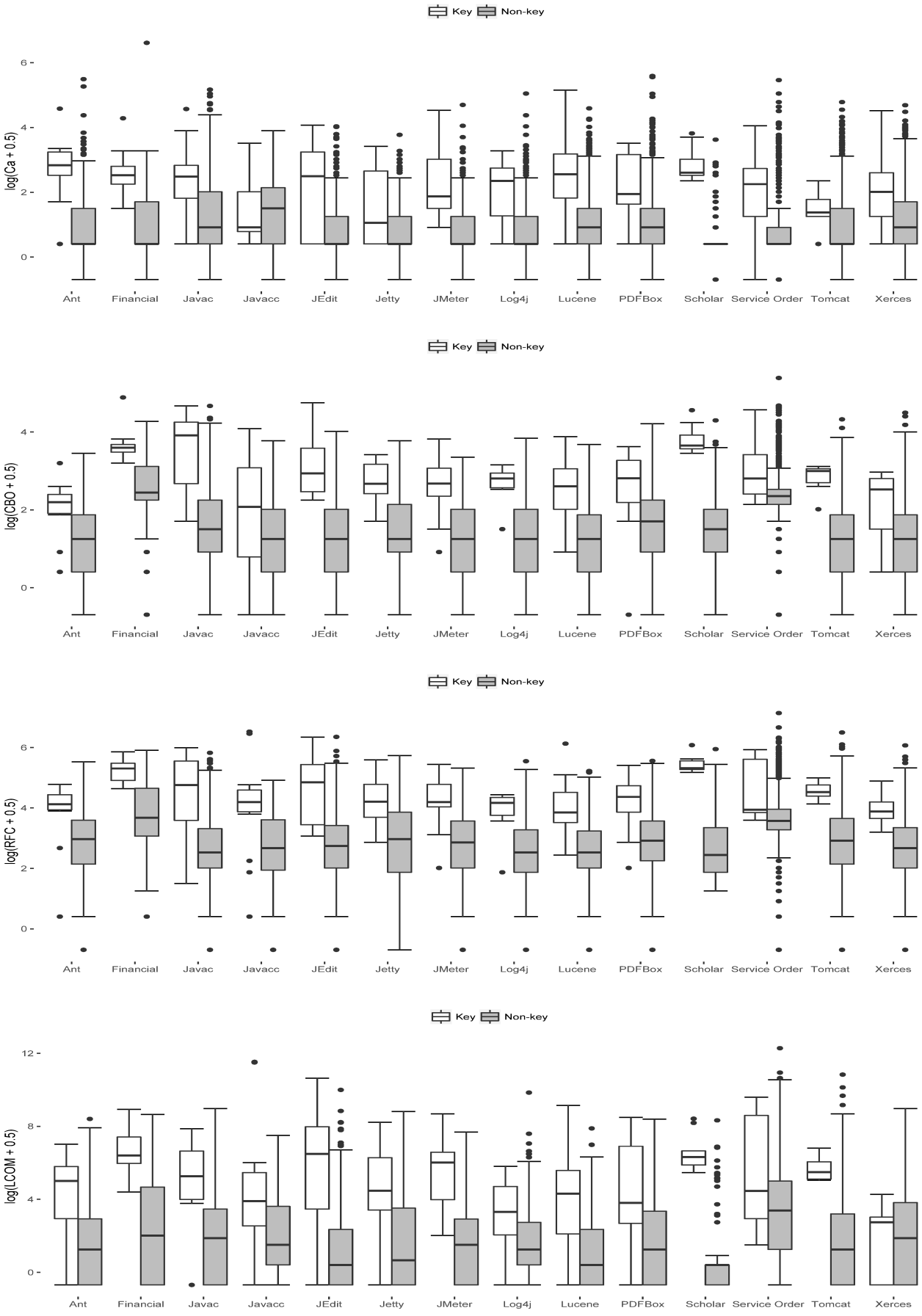


Figure 4 – Metrics CA, CBO, RFC and LCOM in key classes and non-key classes

Table 9 – Statistic Tests between gold set (gs) and key classes (kc) on metrics

	CBO		RFC		LCOM		Ca	
	gs	kc	gs	kc	gs	kc	gs	kc
Min.	0.000	0.000	0.00	0.00	0.00	0.00	0.000	0.000
1st Qu.	2.000	2.000	8.00	8.00	0.00	0.00	1.000	1.000
Median	5.000	5.000	20.00	19.00	5.00	4.00	2.000	1.000
Mean	7.385	7.215	33.98	33.51	233.00	226.06	5.081	4.881
3rd Qu.	9.000	9.000	39.00	38.00	39.00	35.25	4.000	4.000
Max.	218.000	218.000	1260.00	1260.00	215687	215687.00	743.000	743.000
p-value	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$

RQ_3: *What kind of relationship can be found between cohesion/coupling metrics and bad smells?* We want to understand what kind of bad smells are mainly related to cohesion and coupling. For instance, an hypothesis is that a bad smell *large and complex* key class is associated with worse cohesion and coupling. Thus, the complexity associated with these key classes can be related to the fact that classes are more involved in specific smells.

In Table 8 the code smells are basically at three main types: the *LongMethod, LongParamaterList* and *ComplexClass* analyzed in RQ_1. In particular, this is an expected result. Long methods and complex class are related to lower cohesion in classes. Moreover, long parameter lists and complex classes are also related to high coupling. So, addressing those bad smells seems to be a natural way to improve these modularity indicators. Moreover, key classes seems to have higher priority due to its higher impact on the overall design.

Summary of results: Key classes have proportionally more *Complex class, Long Methods and Long Parameter List* smells compared to non-key classes. Also, median values for coupling and cohesion metrics for key classes are significantly worse than for non-key classes. However, there is a significant number of non-key classes with bad smells and poor metrics, so we suggest that prioritizing design assessment based on key classes analysis instead of based on ranked lists of poor-metric classes provides a more focused way to find more relevant design anomalies supported by the design nature of key classes as observed during structural properties analyzes.

4.3 Graph Dependency Assessment

Key classes could be used to enable developers focusing on the design of the target system. An aspect we want to investigate is the possibility of establishing a design view using a dependency graph on key classes, and if this structure is able to reveal important dependencies of source code such as circular dependencies, which are considered problematic (ZIMMERMANN; NAGAPPAN, 2007). More specifically, the study aims at addressing the following question:

RQ_4: *Does a dependency graph of key classes provides a meaningful view of the design?* To tackle this question, we propose to create a dependency graph of key classes and then analyze the degree of adherence between produced output and actual documentation focusing on cir-

cular dependencies to assess design. Moreover, this organization could reveal inconsistencies or provide additional information on the available documentation. The documentation does not necessarily match what is in the source code, either because it shows only a simplified picture, or because it is outdated. The dependency graph of the main classes of the system can display undesirable dependencies. The dependency relationships among key classes are obtained through static analysis. In a static dependency graph, the nodes are classes of the system and the edges indicate their dependencies.

The generation of a dependency graph of key classes is depicted in Figure 5. In that graph, nodes are the classes *A, B, C, D, E, F*. Suppose all nodes are key classes, except node *B*. We remove from the dependency graph the non-key classes, keeping the indirect relationships. When we remove node *B* (see Figure 5 (b)), an indirect relationship of $A \longrightarrow C$ is established. An issue in a dependency graph is the existence of circular relationships, because it may be problematic from the point of view of design. Then, it is important to highlight and understand cycles on dependency graphs, as shown in Figure 5(c) $F \longrightarrow D$.

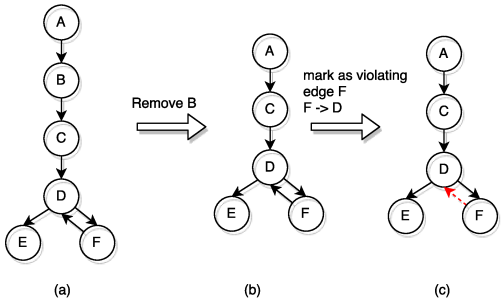


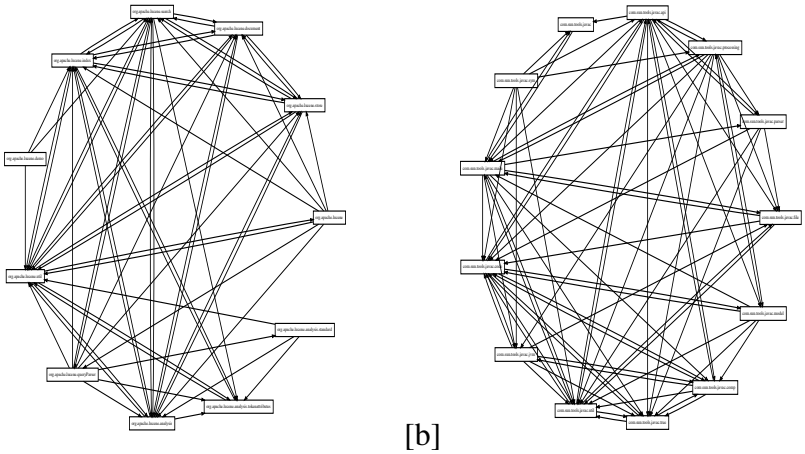
Figure 5 – Dependency graph.

There are some tools that generate dependency analysis and report design violations. Structure 101[4] was reported to have robust features (PRUIJT; KÖPPE; BRINKKEMPER, 2013). So, we adopted Structure101 to extract the dependency graph of all system classes and in sequence, we constructed the dependency graph only to key classes in according to Figure **??**.

To evaluate the meaninfulness of generated organizations, we used Lucene and Javac as baseline because they have a graphical design overview organized in layers (Lucene - Figure 7(a)) and organized in pipes/filters (Javac - Figures 7(b)) and so we can use this baseline for comparison. Initially we obtained the dependency graphs for packages shown in the traces using static analysis shown in the Figure 6 for Lucene and Javac. This figure is intentionally depicted without its full details details as it serves just to show the complexity of coupling and thus motivate the necessity to focus on classes to make dependency analysis more intuitive. Next, we refined the dependency graphs considering key classes recovered by the approach as shown in the Figures 7(c) and 7(d) for Lucene and Javac, respectively. In particular, Keecle

[4] https://structure101.com/

recovers concrete key classes, for Lucene we also considered abstract classes and interfaces which key classes extend or implement because the gold set have those kind of classes. In Javac gold set there were only concrete classes.



[a] [b]

Figure 6 – Dependency graph. a) Dependency graph for Lucene. b) Dependency graph for Javac.

Lucene has two main features: *indexing* and *searching* represented respectively by classes `IndexWriter` and `IndexSearch`. Initially, documents are indexed to a subdirectory after being processed to get indexed tokens. Then, the search for related documents can be performed using queries.

Figure 7(a) represents the layered organization of Lucene dependencies obtained from Lucene's SourceForge repository documentation. This layered design will be used as the baseline for comparison.

Figure 7(c) presents the dependency graph generated from static analysis that shows that the dependency graph provides complementary information to Figure 7(a), because it is more lower-level descending to the level of classes. We can observe that dependencies are implicit in the documentation. So, in this example, the dependency graph will offer an additional support for documentation to make it more clear from the view point of the more dependencies important in terms of key classes, in particular for circular dependency (FSDirectory $\longleftrightarrow$NIOFSDirectory). In situations where the documentation is not available, the automatically produced graph can be useful to understand the organization of the key classes and their dependencies.

Figure 7(b) presents a overview of the compilation process in Javac extracted from documentation. There are three main stages: parsing, processing and generate.

In summary, the six packages performing the main functions are:
- *main* package controlling the code handling process organized into stages.

[8] http://lucene.sourceforge.net/talks/pisa/
[9] http://openjdk.java.net/groups/compiler/doc/compilation-overview/

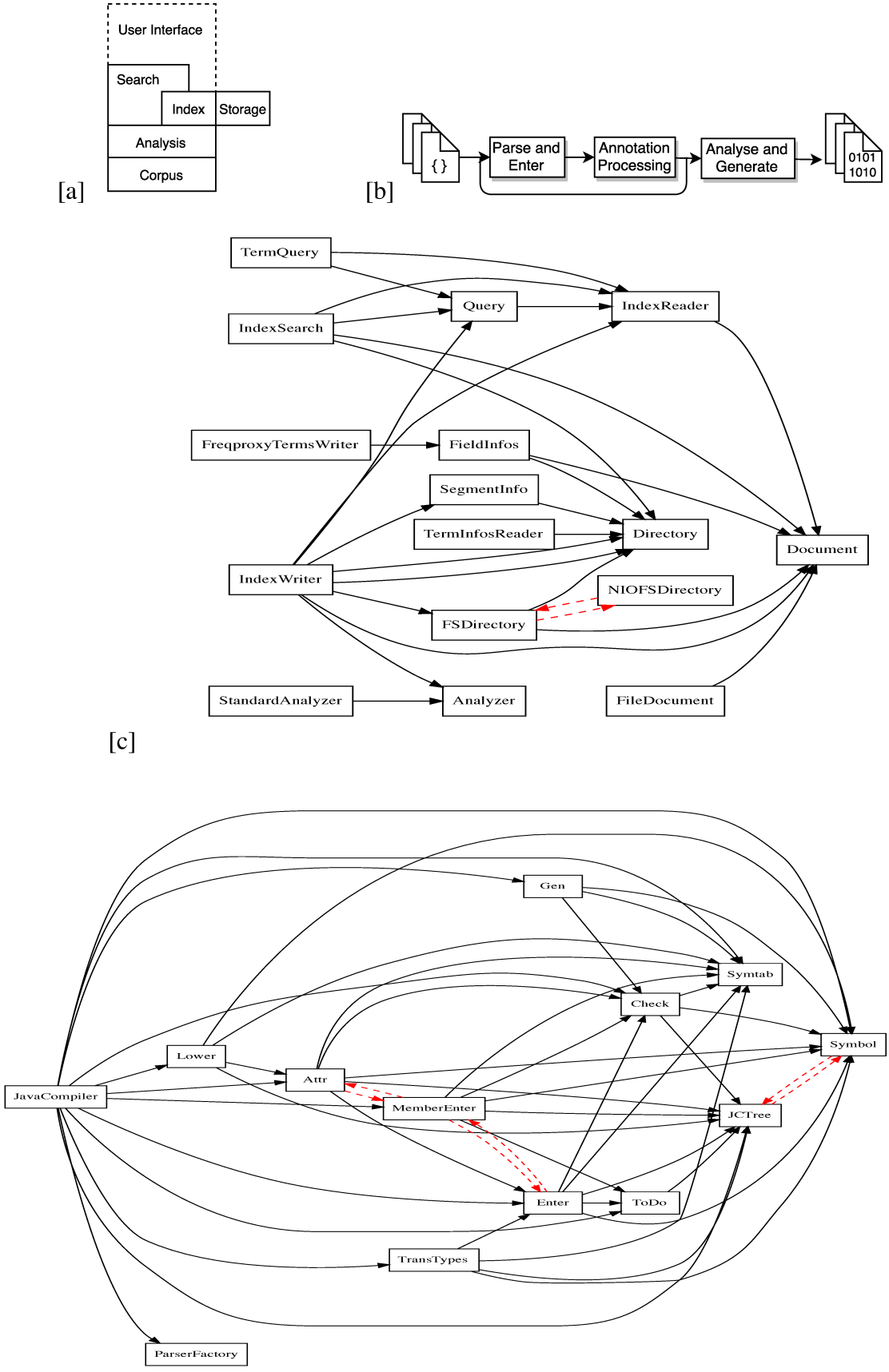


Figure 7 – (a) Lucene Overview.[8] (b) Javac Overview.[9] (c) Static Dependency Graph for Lucene. (d) Static Dependency Graph for Javac.

- *code* package preparing the code files to be read and processed;
- *tree* package responsible for creating syntax trees;
- *comp* package creating class files;
- *jvm* package generating the bytecodes needed by a JVM;
- *parser* package responsible for the parsing algorithm.

In the dependency graph of the Figure 7(d) there are circular dependencies for example: (Attr $\longleftrightarrow$ MemberEnter; MemberEnter $\longleftrightarrow$ Enter and JCTree $\longleftrightarrow$ Symbol). We can observe that the dependencies among the key classes are not as simple as one would expect. Nonetheless, the suggested dependency graph helps to understand the details of dependencies at a still higher level of abstraction.

Summary of results: Dependency graph of key classes is an alternative source of information to assess design because they show the implicit dependencies, show potential inconsistencies between the code and the documentation and also reveal circular dependencies when they exist. Although dependency graph of key classes may be more lower level than human-written architectural documentation, they still are acceptably summarized to be evaluated by the developer.

4.4 Ownership assessment

Human factors have been used to represent an important role in the quality of software components, and thus in design (NAGAPPAN; MURPHY; BASILI, 2008),(PINZGER; NAGAPPAN; MURPHY, 2008). One dimension of human factors is team collaboration with diverse responsibility assignment. Ownership is a property that describes whether one person has responsibility on a software component. Actually, to achieve a concrete proxy for ownership, we consider ownership as the proportion of number of commits in a class, i.e., the main owner is the one with highest number of commits. Under this definition, we may have classes with strong ownership, or classes where ownership is distributed among several developers.

Table 10 – Number of commits in key classes/non-key classes and number of developers.

System	# Commits		# Developers
	kc	nkc	
Ant	935	13220	91
JMeter	1085	10474	43
PDFBox	708	5727	18
Javac	279	2347	437
Lucene	705	48384	144
Jetty	557	11788	157
Tomcat	2707	14295	39
JavaCC	550	186	17
Xerces	2630	2855	26
Log4J	277	7091	60
JEdit	2249	6099	57

Understanding the ownership pattern, if any, depending on the type of class may reveal the level of responsibility placed upon the design of the core developers (GELDENHUYS, 2010).

We rely on the frequency of commits to define the *ownership level* and analyze its relationship to key classes. We also investigate the *degree of heterogeneity*, i.e., if commits in key classes are performed by a higher or lower number of developers compared to non-key classes. To perform this analysis some data was gathered:

(1) The number of commits performed by each developer on classes (ownership level): From the analysis of the results in Figure 8, we observe that ownership of the main developer (the one with the highest number of commits) of key classes is in general lower compared to non-key classes. In fact, the Mann–Whitney–Wilcoxon test showed significant differences between the medians of the two groups ($p < 0.05$). Ant, Jetty, Log4j and Lucene, the number of commits for key classes tends to be less concentrated in a principal developer around 30% of commits and PDFBox and Log4j around 40% of commits. Finally, Javac, JavaCC, JEDit, JMeter and Tomcat in relation to previous groups, the number of commits for key classes tends to be more concentrated in a principal developer around from 60% to 80% of commits. This is an indication that responsibility in key classes tends to be less concentrated in a principal developer. This analysis does not mean that there is not a principal developer that conceived the class structure. The result shows that other developers are also working on key classes instead of leaving the work to just one person.

(2) The number of commits performed only by two main owners: In Table 11 the relative ownership was calculated only with commits performed by the two main owners (we removed the real names of the developers). Noteworthy here is that the top-2 main owners of classes respond for almost all commits. For Lucene, we noted that only two main owners respond for almost 99% of all commits in key classes, whereas the same top-2 owners respond for only 60% of commits in non-key classes. Ant had a similar pattern as Lucene. In JMeter there is sensible difference, and in Javac the main owner in non-key classes responds for almost all commits of main owners. Interestingly, Javac has one additional main owner responding for around 10% of main owner commits in key classes. For other systems we can observe regular distribution of commits. In Tomcat and Log4j the same top-2 owners respond to key classes and non-key classes.

For gold set classes, Javac has similar distribution of commits between two owners for *kc* and *gs* sets. For both *nkc* and *ngs* sets, the distribution is identical. For other systems the situation was similar to key classes.

(3) The number of developers who have committed on each class: Table 10 characterizes the systems in terms of number of commits in key classes/non-key classes and the number of distinct developers. A complementary analysis shown in the Figure 9 indicates that the number of the developers that work in key classes is higher when compared to non-key classes. In fact, this result is consistent with the previous result on ownership: because there are in general more developers working on key classes, the ownership of the main owner of them tends to be lower.

A possible explanation for this is because there are many non-key classes that have less

importance in terms of functionality, and so less developers are “interested” in modifying those classes. Another reason is the higher demand in relation to the key classes and, therefore, a need for collaboration on these classes.

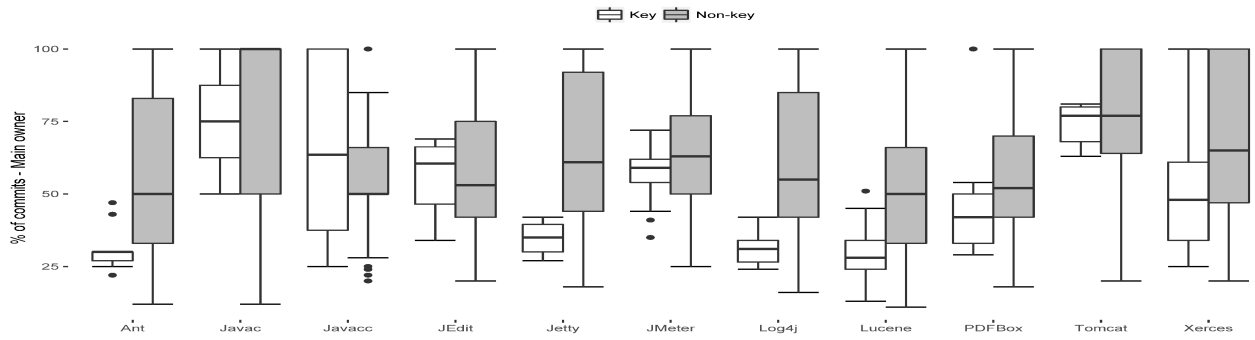


Figure 8 – Relative responsibility (in terms of number of commits) of the main owner of classes

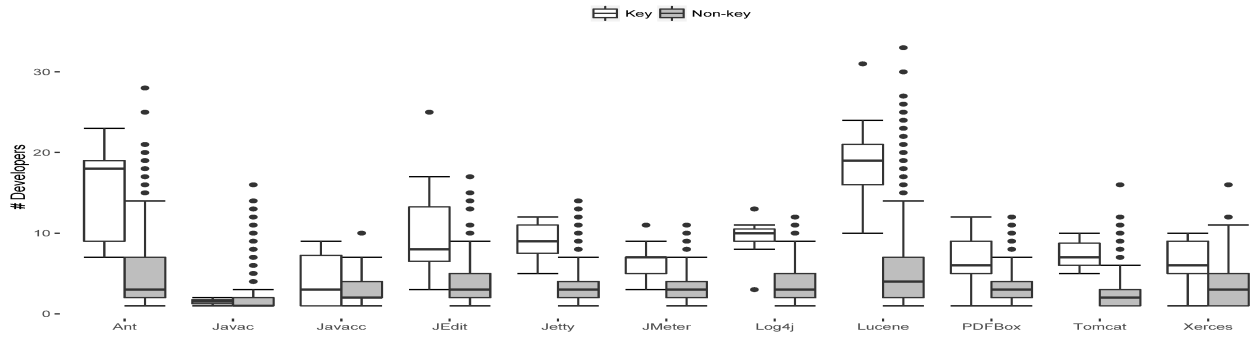


Figure 9 – Number of developers working on classes

We conducted statistical tests as shown in the Table 12 on key classes and gold set, and significant differences among the results are not observed.

Summary of Results: key classes have a lower degree of ownership, being consistent with the higher number of developers, compared to non-key classes. Nonetheless, the top-2 main owners may be more responsible for main owners commits in key classes compared to non-key classes, and so, commits of these top-2 main owners on key classes would have greater impact in design.

4.5 Discussion

The study performed in this chapter between classes recovered using Keecle approach and gold set of the target systems showed that our approach is an alternate way to support non documented or outdated environments because it reported structural and social characteristics with similar results.

Table 11 – % of commits from owners of key classes (kc) and gold set (gs)

Software	Type	Developer	%commits	Software	Type	Developer	%commits
JMeter	kc	SB	98.01	**Xerces**	kc	MGR	29.08
		MS	1.99			HL	18.7
	nkc	SB	94.90		nkc	MGR	28.9
		PM	2.58			SDG	16.17
	gs	SB	97.52		gs	HL	25.6
		MS	2.47			MRG	24.4
	ngs	SB	95.12		ngs	MGR	28.9
		PM	1.89			SDG	15.9
Lucene	kc	MM	53.60	**Tomcat**	kc	MKT	72.9
		RM	45.36			KK	10.7
	nkc	RM	48.20		nkc	MKT	71.3
		MM	13.03			KK	8.7
	gs	MM	89.18		gs	MKT	76.4
		RM	9.05			KK	6.9
	ngs	RM	40.69		ngs	MKT	71.1
		MM	18.18			KK	8.8
Ant	kc	SB	66.43	**JavaCC**	kc	PC	28.9
		PD	28.76			SR	26.9
	nkc	PD	36.50		nkc	PC	32.8
		SB	33.54			SR	20.9
	gs	SB	57.7		gs	PC	32.3
		SB	38.37			TC	20.3
	ngs	PD	35.88		ngs	PC	31.4
		SB	33.65			SR	23.4
Javac	kc	kT	78.60	**PDFBox**	kc	PC	32.8
		MC	11.25			AL	24.62
	nkc	kT	98.84		nkc	TH	27.3
		JJ	0.39			JH	20.2
	gs	kT	84.64		gs	TH	31.7
		MC	8.07			AL	25.9
	ngs	kT	98.84		ngs	TH	27.4
		JJ	0.39			JH	20.2
Jetty	kc	GW	31.53	**Log4j**	kc	GG	25.4
		GWi	25.2			RG	25.11
	nkc	JE	25.8		nkc	GG	30.3
		GW	19			RG	23.3
	gs	GW	30.06		gs	RG	28.2
		GWi	25.04			GG	28.1
	ngs	PD	35.88		ngs	GG	30.22
		GW	19.1			RG	23.11

Despite all classes having a specific role in a system, there are classes that are more important than others, and are more like to have impact in software design. We identified those classes as key classes.

We also have shown that key classes have properties that help developers to assess the overall design and possibly indicate the critical parts of the system that need attention in order to improve that overall design quality. However, we need to mention that the set of key classes recovered of Keecle is not definitive, mainly because of two points. First, the listing of key classes is sensible to target number of key classes k defined by the user. If the user wants to manage more detail, a higher k is selected. On the other hand, a lower k may be selected to assess just a few key classes. The other point is that the set of key classes may vary between different versions of the system. In this study, we applied the approach to a particular version of the target systems, so the set of key classes may not be same compared to other versions

Table 12 – Statistic test between gold set (gs) and key class (kc) on owner and number of developer

	Owner		NDeveloper	
	gs	kc	gs	kc
Min.	11.00	11.00	1.000	1.000
1st Qu.	34.00	50.00	2.000	1.000
Median	50.00	66.00	3.000	2.000
Mean	57.34	67.64	6.958	3.143
3rd Qu.	78.00	100.00	7.000	4.000
Max.	100.00	100.00	34.000	33.000
p-value	$3.511e^{-10}$	$2.2e^{-16}$	$2.2e^{-16}$	$2.2e^{-16}$

because of design evolution. To analyze design evolution based on key classes, different values for *k* should be analyzed based on the assumption that as the size of systems tend to grow so would be the set of key classes.

The fact that key classes tend to present more structural anomalies compared to other classes can be linked to the strong control that those classes have on the application. In general, key classes are able to manage important features of the software, being this fact a possible reason to increase their general complexity, smell occurrence and dependency violations. The latter is observed when we analyze key classes in a dependency graph. In particular, Javac may have presented a higher number of dependency violations because of the large number of key classes. Lucene showed the worst values of LCOM metric, which can be related to the large number of classes containing the *Long Method* smell. On the other hand, in Ant only four smell kinds were detected, which can be related to the low structural complexity identified by the metrics.

A recent study highlighted the importance of producing documentation containing design description on open-source projects and emphasizes the main problems found in the current documentation (ROBILLARD; MEDVIDOVIĊ, 2016). Keecle identifies design key classes, and organize them into dependency graphs and diagnoses the main problems of the class keys. As a motivating example to highlight the importance of that diagnosis, Keecle recovered the *IndexWriter* key class in Lucene. We identified that this class has the worst LCOM value compared to all system classes (LCOM=9414). The RFC value is 458 and the coupling between object classes (CBO metric) is equal to 48. These values indicated that *IndexWriter* is a complex class and because of that, it has the corresponding kind of smells: *ComplexClass, AntiSingleton, ClassDataShouldBePrivate, LongParameterList* and *SpaghettiCode*. We conducted an analysis to associate these problems with discussion threads related to *IndexWriter* available in Lucene issue tracker[5]. We found 65 open issues associated with design problems. We searched for issues related to the terms “refactoring”, “cohesion”, “coupling” and “design” and filtered them with *Unresolved* status since 2009. In particular, we found the (Lucene-2026 “Refactoring of IndexWriter”) issue indicating apply refactoring in *IndexWriter* and several solutions were proposed, but none were was carried out. Refactoring activities is difficult as they can impact other classes (FOWLER et al., 1999)(CHAPARRO et al., 2014). This particular issue for *IndexWriter*

[5] https://issues.apache.org/jira/issues/?jql=project%20%3D%20LUCENE

may remain unresolved, but at least we could quickly report on this situation and the information of dependency of *IndexWriter* with other key classe may prevent other problems being continually inserted.

In contrast, the *QueryParser* class was not selected as a key class by Keecle. Analyzing the structural properties of this class it is the third class with the highest lack of cohesion (LCOM=2658) between all system classes. In sequence, the value to other metrics were: Ca=1; RFC=181; CBO=39. DECOR detected three kind of smells: *BaseClassShouldBeAbstract, ClassDataShouldBePrivate* and *ComplexClass*. Although a class with smells is a problem for the code structure, there are some categories of smells, which are more deeply studied than others and those significantly decrease a class design quality. In this case the most relevant smells in *QueryParser* are *ComplexClass* and *ClassDataShouldBePrivate*. In contrast to *IndexWriter* that has five detected worst smells, four of them critical (*ClassDataShouldBePrivate, ComplexClass, LongParameterList* and *SpaghettiCode*) because they are the most frequent and persistent bad smells) (TUFANO et al., 2015)(CHARALAMPIDOU; AMPATZOGLOU; AVGERIOU, 2015). When we evaluate the issues of *QueryParser* we found only 13 open issues and *Unresolved* resolutions since 2012. In other words, *IndexWriter* seems to have more impact on design compared to *QueryParser*, and thus should require great priority from developers.

Another similar situation involving LCOM metric is related to cohesion deltas. For understanding the evolution aspects, we analyzed PDFBox. We recovered the LCOM of key classes from 12 releases of that software as shown on Figure 10. In particular we can note that LCOM of key classes is noticeably higher in relation to non-key classes for all releases analyzed. Thus, in other words, focusing on key class could drive and reduce developer time to assess the system design and propose new solutions for future releases.

So, the fact that key classes are design classes could redirect in different forms of refactoring in relation to the other classes of the system, in order to reduce the complexity of the key classes in terms of cohesion and coupling. This could help assess the impact of future design decisions during maintenance activities.

On the ownership of classes, Tufano et al showed that those developers that introduce smells are generally the owners of the file and they are more prone to introduce smells when they have higher workloads (TUFANO et al., 2015). The goal of that study was to analyze change history of software projects, with the purpose of investigating when code smells are introduced by developers, and the circumstances and reasons behind smell introduction. Considering we found that the two main owners are responsible for most of the commits of owners in key classes code inspection on commits of these owners would have greater impact on design.

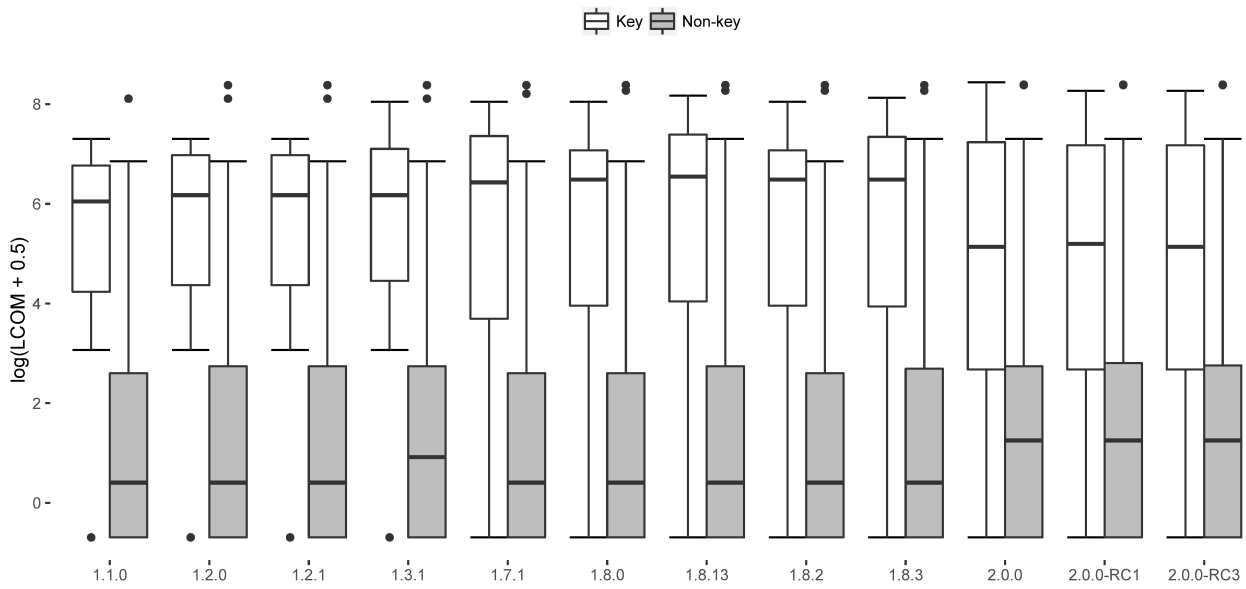


Figure 10 – LCOM Evolution on key classes.

4.5.1 Threats to Validity

Even with the careful planning and formal procedures applied during the execution of the experiments, some threats should be considered in the evaluation of the results validity.

External Validity: There are two points to limit the generalization of our results. Although the systems used in the approach, are well-known systems and used in other studies, factors as the number, size, domain and implementation language (Java) affect the results. Therefore, it is possible that the results will not fully hold for other similar projects, industrial systems or applications developed in other programming languages or paradigms. Other limitation is related to use of only two applications to illustrate organizing on dependency graph, this limit the generalization of our results.

Internal Validity: Different interpretations could occur on key classes from Keecle in relation to the intention of the developer in assuming a class as being a key. Another limitation, DECOR can return classes affected by different types of smells at the same time. But, the interaction between different types of smells was not investigated, because such classes represent a minority (TUFANO et al., 2015).

Construct Validity: We have used an internally constructed tool suite to run Keecle. Although, the tool has been tested and verified, there still may remain some undetected bug as occurs with any software. There is no widely recognized and adopted tool support for this kind of approach, so any other adopted solution would incur a similar threat. We tried to minimize this threat checking the results in each phase. Other problems are due to imprecisions/errors in the measurements we performed. Above all, we used DECOR rules to detect smells, COPE rules

for computing metrics and Structure101 rules for detecting dependencies. We used DECOR, because it was widely evaluated in the literature, with a precision above 60% and a recall of 100% (TUFANO et al., 2015). We are aware that our results can be affected by the presence of false positives and false negatives. Another threat is the analysis of ownership developer, which was performed using the Git author information instead of relying on committers (not all authors have commit privileges in open source projects, hence observing committers would give an imprecise and partial view of the reality). However, there is no guarantee that the reported authorship is always accurate and complete.

4.6 Concluding Remarks

In this chapter, Keecle was applied to open source and proprietary systems to retrieve a reduced number of key classes to investigate the assessment of software design.

Accordingly, due to the design importance associated with those classes, one goal was to investigate structural and ownership properties of key classes compared to non-key classes to analyze the adequacy of using key classes to prioritize design assessment.

The presence of specific bad smells in key classes and the relationship with the metrics of cohesion and coupling were investigated. Our results suggest that that developers should prioritize key classes when assessing design. First, key classes have more *Complex class* smells compared to non-key classes. Second, using conventional structural metrics to prioritize assessment would indicate several non-key classes with poor metrics. Supported by the design nature of key classes, prioritizing design assessment with key classes analysis may increase the chances of finding more relevant design anomalies.

Another analyzed property was related to the dependency graph of key classes. The study showed that the approach produced a meaningful structured view of key classes with respective violations, suggesting that developers would benefit from that in situations where the software documentation is not available, or supplementing current documentation with additional information about dependencies. Finally, on the activity analysis in key classes, we found that although ownership in key classes has a lower level compared to non-key classes, the number of main owners seems to be reduced, either for key and non-key classes, suggesting that prioritizing code review code of owners when committing in key classes would produce more benefits in design.

In the next chapter, we will presented a description of the two experimental studies conducted around key classes. Specifically, the purpose of this study will be to determine whether documentation based on key classes produces a positive feedback on developers.

Chapter 5

Experimental Study with Human Subjects

This chapter contains a description of the two experimental studies conducted around *key classes*. Specifically, the purpose of this research study was to determine whether software documentation based on key classes produces a positive feedback developers.

The major components of this chapter include our research questions and rationale for both quantitative and qualitative study, software system selection, experiment design, data analysis procedures, results, discussion and threats to validity.

5.1 Study Design

In the previous chapter, we have shown that a set of key classes highlights classes that are important from the software design viewpoint, since they present important structural properties, and therefore are able to represent a general organization of the system. In this chapter, we investigate from the point of view of developers if design documentation based on key classes can complement existing documentation or be a replacement for it. The motivation for an experimental study is to investigate if documentation based on key classes would actually benefit comprehension activities, since it is constructed using dynamic analysis, and so, would provide a straight relation to the actual behavior of the software benefiting cognitive activities. Nonetheless, key classes may not cover all details necessary for understanding the systems, and thus additional information would still be necessary.

Another aspect is related to the time. We aim at investigating if a software design documentation based on key classes could help to reduce the time to understand a system. Documentation based on key classes is simple and straightforward, because the set of key classes tends to be small. Therefore, such documentation would help on decreasing system understanding time. The rationale is that a small set of key classes could guide the developer more quickly rather than navigating on all available source files, in case when documentation is not available. On the

other hand, key classes would be more complex than ordinary classes, and still understanding would not be simple.

The experiments had as object of interest, developers during the execution of activities that require analysis of software documentation. Our goal is to present a comparative analysis of different groups about the utility and satisfaction regarding the use of documentation based on key classes compared to the traditional documentation of the target systems. Figure 11 presents an overview experimental study, highlighting the elements that will be discussed on the next sections. Information on the configuration of the experiment and of the survey are described in **Appendix A**.

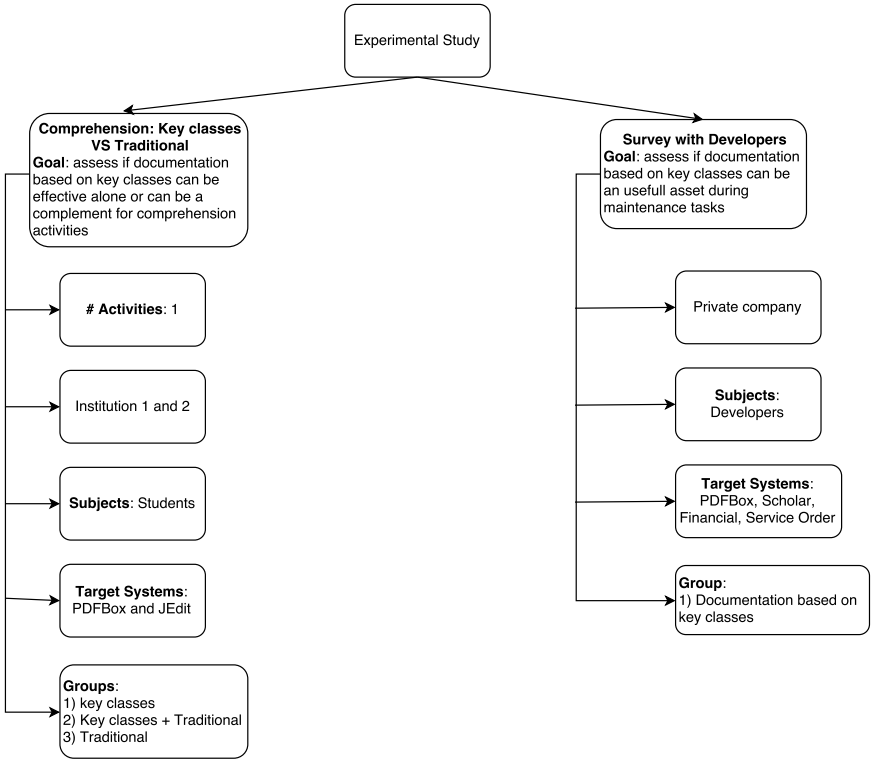


Figure 11 – Experimental Study Overview.

We produced a documentation based on key classes for students and developers, to be evaluated in two experimental studies. In order to define a documentation based on a previous standard, we found that, unfortunately, there is no standard that is widely adopted for developer documentation. A recent work (ROBILLARD; MEDVIDOVIĊ, 2016) highlighted a case study involving the analysis of architectural documentation of 18 source code softwares. In that work, each invited contributor re-documented the architecture of a system on a limited number of pages. Subsequently, the authors of that paper reviewed the documentations and concluded that there was no uniform criterion for documenting a software application.

Instead of delivering for developers a simple list of key classes, we decided to manually produce a documentation based on key classes containing information such as dependency graph between key classes, complexity metrics, code smells, top contributors, trace tree, etc., about

the software systems. One of the reasons for generating information around key classes is based on a study presented in (PINTO; STEINMACHER; GEROSA, 2016) (STEINMACHER et al., 2013), showing that the main barrier for newcomers comprehending a software is the lack of documentation or if it does not contain organized and adequate content. To define the content of the documentation which can help developers to grasp the main aspects of a target system, we performed preliminary meetings with other students and professors. One of the most discussed aspects were the presence of many lines of code in the documentation to comprehend the context. This could be one of the main reasons why newcomers abandon projects.

In this way, we produce documentation that highlights the organization of the system into key classes in a simple and straightforward way, aiming at minimizing the barriers that newcomers face. However, the documentation should also meet the needs of experienced developers, since we highlight structural information about key classes such as code smells and complexity metrics, which may be useful to developers in future software maintenance and evolution activities.

5.2 Experimental Study - Comprehension: Key Classes x Traditional Documentation

In this experiment, the subjects were undergraduate students which would have a profile similar to potential newcomers for an open source systems. We replicated this experiment at two institutions, to investigate if the documentation based on key classes can be used as a starting point for understanding the design of the application for newcomers, compared to the traditional system documentation. Our goal is to assess if documentation based on key classes can be effective alone or can be a complement for comprehension activities. In this experiment, the control factors were the support material that were distributed among the groups.

5.2.1 Study Questions

Based on five criteria – usefulness, time, learning obstacle, satisfaction and easiness for understanding – we establish study questions to verify our hypothesis on key classes:

On the **usefulness** of key classes:

Q1.1: Does the documentation provide a design description that guide the developer in software development activities?

Q1.2: Does the documentation provide useful for understanding the overall organization of the system?

Q1.3: About the information presented in the documentation. What was its usefulness for understanding the application?

Q1.3: Are methods highlighted for specific classes useful for understanding system design?

Q1.4: Was the material provided sufficient and adequate to complete the task?
Q1.5: Was the document navigation mechanism useful for the activity?

On the **time** to evaluate the documentation:
Q2: Does the documentation based on key classes improve the time required to understand an application when compared to the time required in traditional documentation?

On the **learning obstacles** presented by the approach:
Q3: Does the documentation available present obstacles that hinder the learning about the system design by the developer?

On user **satisfaction** observed using documentation based on key classes:
Q4: Is the developer satisfied with completeness and adequacy of the documentation in order?

On the **ease for understanding** of the application using documentation based on key classes:
Q5: Is the documentation easy to understand by the developer?

Next sections, we are going to present experimental setting for this experiment.

5.2.2 Human Subjects

The subjects involved in the experiments are volunteer students not previously involved with the research.

In order to define the subjects and distribute them in groups, we surveyed students for a self-evaluation: i) knowledge level about the object-oriented paradigm; ii) knowledge level about software design; iii) how he/she is classified in relation to the other students in the class.

For the sample selection of students, their teachers of the courses in Java and software design were contacted, and they provided the contact of their students. The participating institutions are listed below:

- Public institution: Bachelor Degree in Computer Science (labeled as Institution 1);
- Public institution: Bachelor Degree in Technology in Systems for Internet (labeled as Institution 2);

The subjects accepted a term of agreement, that stated the conditions of the experiment, including that their identify would not be disclosed. At the end of the experiment the students received an online questionnaire to assess whether the same were able to participate according to inclusion and exclusion criteria of the experiments, described bellow. Thus, 36 students from Institution 1 and 29 students from Institution 2 volunteered to participate.

5.2.3 Experimental Activity

This experiment consisted in performing a design comprehension activity evaluating the documentations (traditional or based on key classes) with students from two different institutions to evaluate the design documentation.

The students from institution 1 analyzed PDFBox documentation and from institution 2 analyzed JEdit documentation. In each institution, students were randomly divided into 3 groups to available documentation for the comprehension activity. So, the activity of this experiment was to comprehend the application design from the available documentation analysis.

- **Traditional - Traditional documentation only group:** evaluated the traditional documentation, available on the developer's website, during the comprehension activity;
- **Key classes + Traditional documentation - complementary group:** evaluated the traditional documentation, available on the developer's website and documentation based on key classes, during the comprehension activity.
- **Key classes only - Documentation based on key classes group:** evaluated the documentation based on key classes, during the comprehension activity;

Regarding to the activity complexity, one possible solution would be to read the content provided in the documentation to locate important components of the application design.

The comprehension activity is designed to be completed in a maximum time limited to 60 minutes for each activity to prevent students spend all the time available to solve the task because at the final of task the students were solicited to fill out a survey. The experiment was conducted in a single day because there was no alternation of documentation between groups. The activities were applied in the computer lab during the class period of the students to guarantee a more participants.

5.2.4 Sample Characteristics

As an inclusion criteria, students were invited to attend if the following criteria were met:

- undergraduate students for bachelor's degree in computer science and bachelor's degree in Technology in Systems for Internet;
- Students currently involved in courses that include knowledge about the object-oriented paradigm and software design.

A questionnaire, was applied to establish a profile of the volunteer. Considering students from Institutions 1 and 2:

Table 13 – Profile of subjects - Institutions 1 and 2.

Institution 1				Institution 2			
Object-Oriented Programming				Object-Oriented Programming			
	Tradit.	**Tradit. + Key classes**	**Key classes**		**Tradit.**	**Tradit. + Key classes**	**Key classes**
Min	5	1	3	**Min**	1	6	5
Max	10	10	8	**Max**	10	9	9
Mean	6.9	6.6	6.4	**Mean**	5	6.9	7
Median	8	7	8	**Median**	5	7	7
Software Design				**Software Design**			
	Tradit.	**Tradit. + Key classes**	**Key classes**		**Tradit.**	**Tradit. + Key classes**	**Key classes**
Min	2	0	0	**Min**	1	2	1
Max	8	9	9	**Max**	9	9	8
Mean	4.1	4.7	5.3	**Mean**	5	4.9	6.2
Median	4	4	5	**Median**	6	6	6
Compared Knowledge Level				**Compared Knowledge Level**			
	Tradit.	**Tradit. + Key classes**	**Key classes**		**Tradit.**	**Tradit. + Key classes**	**Key classes**
Min	5	5	4	**Min**	1	5	5
Max	9	9	8	**Max**	9	9	9
Mean	7.3	7	6.6	**Mean**	6	8	7.1
Median	7	7	7	**Median**	6	8	7

- All are enrolled Computer Science students and were attending or had already attended courses related to the object-oriented paradigm - Institution 1;
- All are enrolled for Technology in Systems for Internet and were attending or had already attended courses related to the object-oriented paradigm - Institution 2;
- On the knowledge level related to the object oriented paradigm informed by the subjects themselves (an auto-evaluation). Using a scale from 0 to 10 we obtain the following results as observed in the Table 13 in terms of minimum level = min; maximum level= max; mean and median.
- On the knowledge level in software design attributed by the subjects themselves. Using a scale from 0 to 10, we obtain the following results as observed in the Table 13 in terms of minimum level = min; maximum level= max; mean and median
- On the knowledge level classification compared to the other students in the class. Using a scale from 0 to 10 we obtain the following results as observed in the Table 13 in terms of minimum level = min; maximum level= max; mean and median.

In relation to the student sample from the two considered institutions, they demonstrate a reasonable knowledge about the object-oriented paradigm and software design.

5.2.5 Target systems

Two Java applications in well-known domains are our objects of study. We suppose that a well-known domain should not be a factor that could interfere substantially in the performance of the subjects. We mined key classes on the target systems and then we define some of the comprehension questions to assess the quality of the recovered key classes.

Apache PDFBox Software

The Apache PDFBox library is an open source Java tool for working with PDF documents. This project allows creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents. Apache PDFBox also includes several command-line utilities.

Moreover, we considered this application because, PDFBox is an adequate representative of real-life programs, having 166 classes distributed across 110 packages, containing the total of 116464 LOC. It also has good documentation and version control, uses examples that help developers to use the application. The source code of the application is made available and finally, the application domain is easy to understand for potential newcomers.

In order to run Keecle to detect the key classes, we considered 13 features obtained from the present examples in the application source code. We then produced a documentation based on key classes and contacted the real developer of the application to evaluate the approach. The documentation based on key classes for PDFBox and it is available in the link of the LASCAM website[1].

The traditional PDFBox documentation, has the following structure:

- **Overview:** this topic contains help, features ans news about PFFBox;
- **License:** this topic contains licensing of distributions;
- **Downloads:** this topic contains information about releases, mirrors, libraries, etc;
- **Support:** this topic contains questions about how to use PDFBox;
- **Mailing lists:** this topic contains questions about or problems with Apache PDFBox;
- **Issue tracker:** this topic contains a knowledge base containing information on each customer, resolutions to common problems, and other such data.
- **Project team:** list of developers with commit privileges that have directly contributed to the project in one way or another;
- **Migration Guide:** this topic contains information about environment required;
- **Examples:** this topic highlights a list of examples which evaluable on SVN[2] (a Version Control System);
- **Dependencies:** this topic contains APIs required to run PDFBox;
- **Building from source code:** this topic contains several ways for building the application;
- **Coding conventions:** this topic contains some rules for formatting text, white space, structure, etc.

[1] http://lascam.facom.ufu.br/pdfbox/PDFBox/theme/
[2] http://subversion.apache.org/

JEdit Software

JEdit is a mature programmer's text editor with the following features:

- Extensibility in which plugins can turn jEdit into a very advanced XML/HTML editor, or a full-fledged IDE, with compiler, code completion, context-sensitive help, debugging, Visual diff, and many language-specific tools tightly integrated with the editor.
- Customization File Management Search and Replace Source Code Editing General Multiple open windows, Unlimited undo / redo, copy and paste, Marker locations.

Among the factors that allowed the choice of this application include:

- JEdit has been addressed in several related works;
- Software has good online documentation and control versions;
- The application domain is easy to understand;

In order to mine the key classes, ten more relevant features of JEdit were exercised, these features were selected by the largest number of classes that were captured by the trace extractor. In relation to traditional documentation of JEdit this presents the following information:

- Features: A detailed view on the application's features;
- Compatibility: Provides information on operating systems and recommended Java versions;
- Reviews: A list of reviews already performed on JEdit;
- Downloads and plugins: this topic contains instructions for installing JEdit and available plugins and how to use them;
- JavaDoc about application classes.

Next, we produced a documentation based on key classes for JEdit. The documentation based on key classes is available in the site of the LASCAM research group[3].

5.2.6 Variables

The experiments are limited by a set of independent and dependent variables which will be presented below. To answer the questions in Table 14 the **independent variable** is the kind of available documentation:

- traditional documentation, documentation based on key-class, or both.

[3] http://lascam.facom.ufu.br/jedit/jEdit/theme/

Student groups were divided into three groups: a control group (traditional documentation), the key classes group and the complementary group (using traditional documentation plus documentation based on key classes). This criterion allows comparing the performance of the proposed approach.

The dependent variables are the answers provided by the questions from the students. The questions from Table 14 were used as affirmative sentences because their answers were measured using the Likert scale:

- 2: Strongly Agree
- 1: Agree
- 0: Neutral - I did not know opine
- -1: Disagree
- -2: Strongly Disagree

Table 14 – Research Questions for Students.

Research Question	
Q.1.1	Is the documentation provided a design documentation that guide the developer in software development activities?
Q1.2	Is the documentation provided useful for understanding the overall organization of the system by the developer?
Q1.3	Are the methods highlighted during the definition of some application classes useful for understanding the system design?
Q1.4	Are the description about application dependencies clear and useful for understanding the application?
Q1.5	Was the document navigation mechanism useful for the activity?
Q2.1	Check below the start and end time of the activity.
Q3.1	Does the documentation available presents problems that limit the learning about the system design by the developer?
Q4.1	Was the material provided sufficient and adequate to complete the task?
Q5.1	Is the documentation easy to understand by the developer?

5.2.7 Data Analysis Methodology

We applied the Mann-Whitney test differences on the median of two groups. The Kruskal-Wallis tests were considered when three groups a compared as, described in sequence.

5.2.7.1 Kruskal-Wallis Test

It is equivalent to the non-parametric ANOVA, where the measured variable must be numerical or ordinal scale and assumptions of normality and homogeneity of variance compromised.

The Kruskal-Wallis test (KRUSKAL; WALLIS,) is used to test the hypothesis that several samples (two or more) have the same distribution.

To interpret a Kruskal-Wallis test, key output includes the point estimates and the p-value. To determine whether any of the differences between the medians are statistically significant, we compare the p-value to the significance level to confirm/reject the null hypothesis. The

null hypothesis states that the population medians are all equal. Usually, a significance level (denoted as a) of 0.05 works well. A significance level of 0.05 indicates a 5% risk of concluding that a difference exists when there is no actual difference. P-value $\leq a$, the differences between some of the medians are statistically significant and P-value $> a$, the differences between the medians are not statistically significant.

5.2.7.2 Mann-Whitney test

The Mann-Whitney U test is often considered the nonparametric alternative to the independent t-test. The Mann-Whitney test is used to compare differences between two independent groups when the dependent variable is either ordinal or continuous, but not normally distributed.

Similar to Kruskal-Wallis test, to determine whether the difference between the medians is statistically significant, we can compare the p-value to the significance level. Usually, a significance level a of 0.05 works well. A significance level of 0.05 indicates a 5% risk of concluding that a difference exists when there is no actual difference. Thus, P-value $\leq a$ the difference between the medians is statistically significant and P-value $> a$, the difference between the medians is not statistically significant.

5.2.8 Results

In this section the results of the individual questions will be presented for an accurate analysis of the strengths and weaknesses of the approach. The analysis will be presented by experiment, and then we will perform a collective analysis of the data.

5.2.8.1 Results - Institution 1

Next, the results will be presented for the experiment performed with 36 students from institution 1 using the Apache PDFBox application. The test results are summarized in the Table 15 and boxplots are presented on the Figure 12 and will be discussed on the next topics. With the exception of the time other research questions were answered through a questionnaire that the subjects answered after the activity. The questions from Table 14 were used as affirmative sentences because their answers were measured using the Likert scale.

Table 15 – Summarized Results of the Experiment - Institution 1.

Criterion	Kruskal-Wallis chi-squared	p-value	Post-Hoc analysis (Traditional / key-classes + Traditional)	Post-Hoc analysis (Traditional/Key-classes)	Post-Hoc analysis (Key-classes/ key-classes + Traditional)
Time	2.4387	0.2954	0.73	0.70	0.26
Navigation Utility	0.18325	0.9124	0.99	0.93	0.98
Documentation Satisfaction	3.4984	0.1739	0.36	0.30	0.99
Method Utility	2.5396	0.2813	0.39	1.00	0.43
Learning Obstacle	4.4508	0.108	0.33	0.13	0.88
Dependencies Utility	0.80561	0.6684	0.77	0.90	0.97
Comprehension Facility	2.5879	0.2742	0.44	1.00	0.41
Usefulness Information	2.8628	0.239	0.30	0.82	0.66

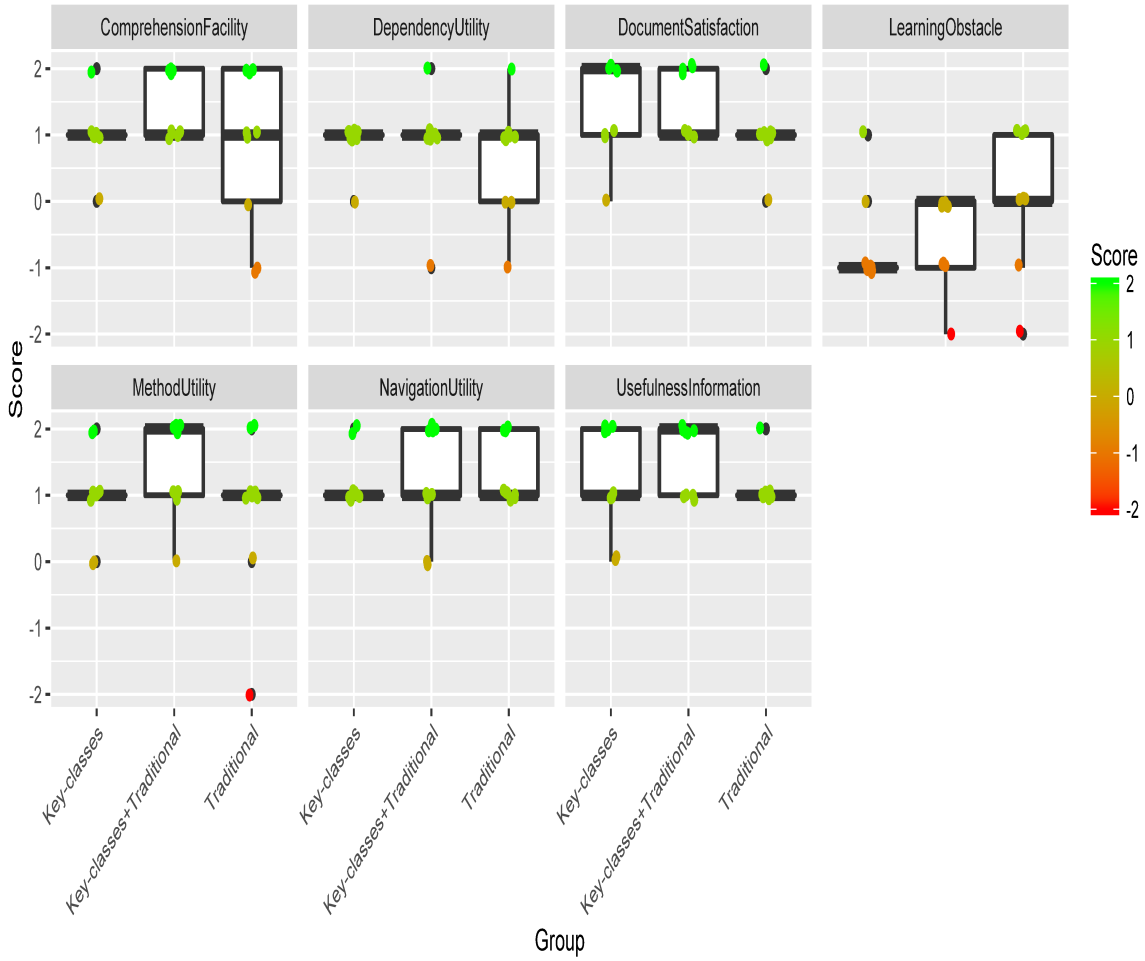


Figure 12 – Boxplot Results of the Experiment - Institution 1.

Time: The time to perform the activity was collected for each subject, which was delimited by a time of 60 minutes per activity. As can be seen in Figure 13 the boxplot of the distribution of the three groups no significant differences in distribution, given the premise for Kruskal-Wallis test. The test result showed a p-value of 0.2954, there was not a significant difference between the groups. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. In a qualitative analysis, group Key-classes presented a shorter time, as seen in the boxplots, although it was not statistically significant.

Navigation Utility: The subjects were asked about the navigation mechanism in the documentation - *The document navigation mechanism was useful for the activity*. The result for the Kruskal-Wallis test presented a p-value of 0.9124. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. Therefore, the perception of navigation mechanism utility had no significant difference between groups. In the boxplot of the Figure 12, it can be noted that the medians are equals one for all groups. So, the subjects of this group, in fact had a similar agreement on the navigation mechanism utility.

Documentation Satisfaction: The subjects were asked about satisfaction about the use of

Figure 13 – Time to Perform the Activities - Institution 1.

the documentation - *The material provided was sufficient and adequate to complete the task.* The result for the Kruskal-Wallis test had a p-value of 0.1739. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. Therefore, the satisfaction perception with the use of the documentation was not significantly different for the three groups. In the boxplot of the Figure 12 we observe for group Key-classes a higher median compared to the other groups. So, the students in group *Key-classes* presented a slightly greater satisfaction regarding the content of the documentation based on key classes, although not statistically significant.

Method Utility: The subjects were asked whether the methods highlighted during the definition of some classes are useful for system design comprehension - *Methods highlighted during the definition of some classes are useful for understanding system design.* The result for the Kruskal-Wallis test presented a p-value of 0.2813. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15, so the methods' utility with the use of the documentation was not significantly different for the three groups. In the boxplot of the Figure 12, we observe for groups Traditional and *Key-classes* a similar median and for group *Key-classes+Traditional* a higher median. So, the both documentation based on key classes and traditional documentation used by group *Key-classes+Traditional* presents information regarding important methods of the application, although not statistically significant.

Learning Obstacle: The subjects were asked if the available documentation presents problems that limit the learning about the system design - *The available documentation present problems that limit the learning about the system design by the developer.* The result for the Kruskal-Wallis test had a p-value of 0.108. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. Therefore, it can be said that the learning limit the use of the documentation was not significantly different for the three groups. In the boxplot of the Figure 12, we can see a lower median for *Key-classes* group

(equals -1) than the other groups, that is, documentation based on key classes presents fewer problems than traditional documentation.

Dependencies Utility: Subjects were asked whether the application dependencies description are clear and useful for understanding of the application - *The description about application dependencies are clear and useful for understanding the application*. The result for the Kruskal-Wallis test presented a p-value of 0.6684. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. Therefore, the dependencies utility with the use of the documentation was not significantly different for the three groups. In the boxplot of the Figure 12 it is noted for the groups *Traditional*, *Key-classes+Traditional* and *Key-classes* also presented similar medians. So this suggest that all types all documentation are similarly adequate to satisfy the comprehension of the developer.

Comprehension Facility: The subjects were asked if the documentation is easy to understand by the developer - *The documentation is easy to understand by the developer*. The result for the Kruskal-Wallis test had a p-value of 0.2742 no significant difference was found, as shown in the Table 15. Therefore, the ease for understanding with the use of the documentation was not significantly different for the three groups. In the boxplot of the Figure 12 for groups *Traditional*, *Key-classes+Traditional* and *Key-classes* also presented similar medians equals one for all groups, suggesting that all types of documentation are similarly adequate to facilitate the understanding of the application design.

Usefulness Information: The subjects were asked about the useful information contained in the two documentations (based on key classes and traditional) to perform the comprehension activity - *The documentation provided is a project documentation that guide the developer in software development activities.* They responded according to the boxplots of the Figure 12. The result for the Kruskal-Wallis test showed a p-value of 0.239, showing that there was no significant difference in the perception of the documentation usefulness. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 15. Analyzing the medians on the boxplots, it is observed that the group *Key-classes+Traditional*, which performed the evaluation of the two documentations (based on key classes and traditional), presented a perception of greater utility. This observation suggests that the two documentation are complementary to each other.

5.2.8.2 Results - Institution 2

Next, the results for the experiment performed with 29 students from institution 2, using the JEDit application. The test results are summarized in the Table 16 and boxplots are presented on the Figure 14. With the exception of the *Time* other research questions were answered through a questionnaire that the subjects answered after each activity. The questions from Table 14 were used as affirmative sentences because their answers were measured using the Likert scale.

Table 16 – Summarized Results of the Experiment-Institution 2.

Criterion	Kruskal-Wallis chi-squared	p-value	Post-Hoc analysis (Traditional / Key-classes + Traditional)	Post-Hoc analysis (Traditional / Key-classes)	Post-Hoc analysis (Key-classes / Key-classes + Traditional)
Time	1.8732	0.392	0.98	0.98	0.93
Navigation Utility	8.3319	**0.01552**	0.957	0.097	0.050
Documentation Satisfaction	2.8276	0.2432	0.37	0.82	0.74
Methods Utility	1.5447	0.4619	0.93	0.51	0.75
Learning Obstacle	0.4112	0.8142	1.00	0.85	0.85
Dependencies Utility	1.1473	0.5635	0.97	0.76	0.62
Comprehension Facility	3.872	0.1443	0.26	1.00	0.25
Usefulness Information	0.16815	0.9194	0.99	0.98	0.94

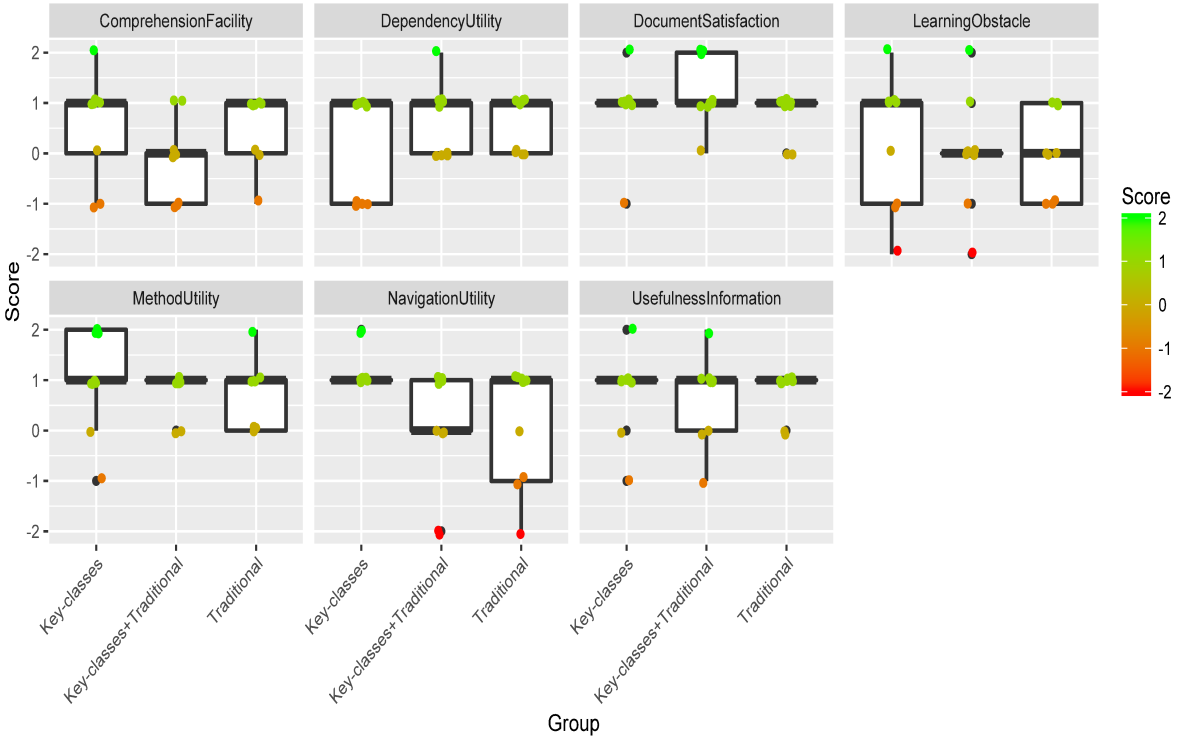


Figure 14 – Boxplot Results of the Experiment-Institution 2.

Time: The time to perform the activity was collected for each subject, which was delimited by a time of 60 minutes per activity. Figure 15 shows the boxplot of the distribution of the two groups no significant differences in distribution, shape satisfying the premise for Kruskal-Wallis test. The test result showed a p-value of 0.392 which is higher considering a commonly adopted standard, it can not be said that there was a significant difference between the groups. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 16. In a qualitative analysis, group *Key-classes* presented a smaller amount of time related results, as observed in the boxplot.

Navigation Utility: The subjects were asked about the navigation mechanism in the documentation - *The document navigation mechanism was useful for the activity*. The result for the Kruskal-Wallis test had a p-value of 0.01552, lower than the significance level. Therefore, it can be said that the perception of navigation mechanism utility had a significant difference between

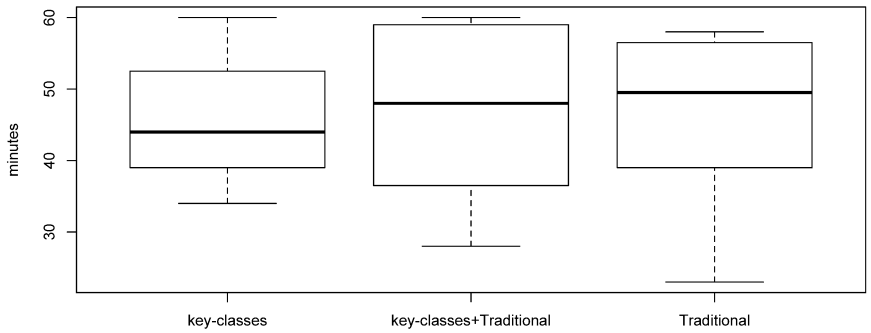


Figure 15 – Time to Perform the Activities - Institution 2

the groups. When we performed Post-Hoc analysis on the groups, we found a significant difference between *Key classes - Key-classes+Traditional* groups (equals to 0.050). In the boxplot of Figure 14, it can be observed that most of the students in Group *Key-classes* presented regular agreement on the navigation mechanism utility.

Documentation Satisfaction: The subjects were asked about satisfaction with the use of the documentation - *The material provided was sufficient and adequate to complete the task.* The result for the Kruskal-Wallis test had a p-value of 0.2432. When we performed Post-Hoc analysis on the groups, we also did not find significant difference. Boxplot in Figure 14 note by the analysis of the medians that the three groups had similar satisfaction equals one, regarding the contents of the documentation based on key classes.

Methods Utility: The subjects were asked whether the methods highlighted during the definition of some application classes are useful for system design comprehension - *Methods highlighted during the definition of some classes are useful for understanding system design.* The result for the Kruskal-Wallis test had a p-value of 0.4619. When we performed Post-Hoc analysis on the groups, we also did not find significant difference. In the Figure 14 boxplot groups *Traditional*, *Key-classes+Traditional* and *Key-classes* presented similar medians equals one indicating that the all kinds of documentation presented important information to methods.

Learning Obstacle: The subjects were asked if the available documentation presents problems that limit the learning about the system design - *The documentation available present problems that limit the learning about the system design by the developer.* The result for the Kruskal-Wallis test had a p-value of 0.8142. When we performed Post-Hoc analysis on the groups, we also did not find significant difference. Boxplot in Figure 14 note to the group *Key-classes* a higher median equals one, compared to the groups *Traditional* and *Key-classes+Traditional*. So, documentation based on key class-based presents major problems which limit the learning regarding traditional documentation. Possible problems are related to bad smells, trace trees concepts, etc., which are not common terms to students.

Dependencies Utility: Subjects were asked whether the description of application dependencies are clear and useful for comprehending the application - *The description about application dependencies are clear and useful for understanding the application*. The result for the Kruskal-Wallis test showed a p-value of 0.5635. When we performed Post-Hoc analysis on the groups, we also did not find significant difference. In the boxplot of the Figure 14 it is noted for the groups *Traditional*, *Key-classes+Traditional* and *Key-classes* similar medians equals one. So, information about the dependencies contained in the documentation based on key classes and the traditional adequately guide the subjects to the information comprehension.

Comprehension Facility: The subjects were asked if the documentation is easy to understand by the developer - *The documentation is easy to understand by the developer*. The result for the Kruskal-Wallis test was a p-value of 0.1443. When we performed Post-Hoc analysis on the groups, we also did not find significant difference. In the boxplot of the Figure 14, for groups *Traditional* and *Key-classes* presented similar medians equals one. So, information is organized in the documentation based on key classes and the traditional facilitate the design comprehension.

Usefulness Information: The subjects were asked about an information utility contained in the two documentations (based on key classes and traditional) to perform the activities - *The documentation provided is a project documentation that guide the developer in software development activities.* They answered according to the boxplot of the Figure 14. The result for the Kruskal-Wallis test presented a p-value of 0.9194. When we performed Post-Hoc analysis on the groups, we did not find significant difference as shown on the Table 16. From the median of the boxplots, that the information available for all groups have median equals one suggesting were equally, and suggests that documentation based on key classes can be used in environments where no documentation is available or outdated and it can complement traditional documentation.

5.3 Survey with Developers

In this study, the subjects were developers, who analyzed if the documentation based on key classes could replace the traditional documentation and whether the information contained in the documentation could support the developers in maintenance activities.

For the developers sample, approximately 29 developers of open source systems and property systems were invited to participate. Four developers accepted to participate were three developers from a private company and one developer was from an open source system. One of these subjects was acknowledge of the researchers characterizing a possible thread for the research. To mitigate this threat the research questions were not subjective and involved a real evaluation activity of the documentation based on key class. In addition, there were control questions to discard inappropriate subject answers from the analysis.

5.3.1 Questions of the Survey

Based primarily on six criteria based on quality and utility, the following questions are formulated to investigate our hypothesis on key classes:

Documentation Quality:

Q1.1: Are Dependency graphs important for understanding the main dependencies of the application?

Q1.2: Are actually key classes important entities for design/architecture comprehension?

Q1.3: Is this new documentation a design documentation?

Q1.4: Is the provided information useful to complement the understanding of the general organization of the system?

Q1.5: Can this new documentation replace the traditional documentation of the original developer used in application?

Q1.6: Is the new documentation is easy to understand?

Q1.7: Is there missing information in the generated documentation?

Key classes Quality:

Q2.1: Are key classes set are enough for evaluating architecture of the application?

Q2.2: Are key classes set adequate starting point to comprehend the application?

Q2.3: Are knowing key classes useful information for software maintenance?

Q2.4: Are knowing key classes useful information for introduction of new functionalities?

Q2.5: Are knowing key classes useful information for bug fixing?

Q2.6: Could if your system has some architectural problems be solved restructuring key classes?

Smell Detection Utility:

Q3.1: Are detected smells presented in the documentation are useful to show design anomalies?

Q3.2: Are knowing detected smells useful information for software maintenance?

Q3.3: Are knowing detected smells useful information for introduction of new functionalities?

Q3.4: Are knowing detected smells useful information for bug fixing?

Trace Tree Utility:

Q4.1: Are the trace trees presented in the documentation useful information to show design anomalies?

Q4.2: Are knowing trace trees useful information for software maintenance?

Q4.3: Are knowing trace trees useful information for new functionalities?

Q4.4: Are knowing trace trees useful information for bug fixing?

Q4.5: Are the Graphs "All usages" presented in the documentation useful to show design anomalies?

Q4.6: Are the methods and attributes presented in the documentation useful for software maintenance or introduction of new functionalities?

Dependency Graph Utility:

Q5.1: Are knowing dependency graph useful information for bug fixing?

Q5.2: Are knowing dependency graph useful information for software maintenance?

Q5.3: Are knowing dependency graph useful information for introduction of new functionalities?

Q5.4: Are knowing dependency graph useful information for detecting smells?

Complexity Metrics Utility:

Q6.1: Considering complexity metrics boxplots. Are boxplots set enough for evaluating architectural problems of the application?

Q6.1: Are knowing complexity metrics useful information for introduction of new functionalities?

Q6.1: Are knowing complexity metrics useful information for software maintenance?

Q6.1: Are knowing complexity metrics useful information for bug fixing?

5.3.2 Sample Characteristics and Inclusion Criterion

Regarding to the inclusion criteria to invite developer to participate in the survey were established two criteria:

- Developers with experience in the Java programming language;
- Experience and knowledge level in the target application.

In sequence, developers, they were asked to report their experience level to identify possible newcomers. All of them reported have a professional experience that goes to from years (from 5 to 11 years), not being newcomers and have the owner profile of the application. About the open source application developer we checked his information available in OpenHub and this is a contributor around 8 years and is the manager of the application.

5.3.3 Target systems

Four Java proprietary applications will be our objects of study. We mined key classes on target systems and then we define some of the comprehension questions to assess the quality of the recovered key classes and the application design comprehension of them. Proprietary systems or open source which did not presented evaluable documentation or did not have documentation based on key classes are adequate to show the feasibility of our approach in an industrial context.

Table 17 reports for each of such (a) Running Scenario (b) the number of lines of code, (c) the number of packages, and (d) the number of classes. The choice of the systems for analyzing was driven by distinct size, design and application domain and the developer's interest

in collaborating with the evaluation of the approach. In sequence will be present the main features of target systems, PDFBox was previously discussed.

Table 17 – Characteristics of the systems under analysis.

System	# Scenarios	LOC	Packages	Classes
Financial	9	36702	21	130
School	11	59427	40	424
Service order	14	558534	183	3361

Scholar Software

The Scholar is a proprietary system, is destined to regular schools from small to large size, developed with Java technology.

In order to mine the key classes, we invited its developers to collaborate, so the ten more relevant features of Scholar were pointed out by the owner of the application to exercise them. This application does not have evaluable traditional documentation so, a documentation based on key classes was produced. The executed features of the Scholar software are listed in sequence:

- Maintenance of the Bulletin (Release of notes, faults);
- Issuance of the School Report Card;
- Issuance of School Records;
- Registration of goods for maintenance;
- Disciplines control;
- School management;
- Parents and students control;
- Matrix Control of Disciplines;
- Occurrences control of students.

Financial Software

This application is developed in Java language and can register the price tables by modalities of courses and can include additional rates and amounts according to the financial policy of the institution. Based on this price list are created payment plans that are flexible to the needs of the institution.

In order to mine the key classes, we invited its developers to collaborate, so the ten more relevant features of Financial were pointed out by owner of the application. This application does not have available traditional documentation. So, a documentation based on key classes was produced. The executed features of Financial are listed in sequence:

- Management Tuition;
- Employee Control;
- Enrollment payment management;
- Management of Purchase of school material;
- Control of Sale of school material;
- Management of accounts payable and receivable;
- Cash Flow Control;
- Issuance of bank tickets;
- Issuance of individual and grouped containers;
- Control of pre-dated checks;
- Financial Reports.

Service Order Software

Service Order software is developed in Java technology and allows to link strategic and operational information with various activities related to providing services.

In order to mine the key classes, we invited its developers to collaborate, so the ten more relevant features of Service Order were pointed out by owner of the application. This application does not have available traditional documentation, a documentation based on key classes was produced. In relation to features exercised of Service Order are listed in sequence:

- Multi company;
- Print the Service Order in several formats, including PDF;
- Allows inform the participant technical of the maintenance;
- Registration of goods for maintenance;
- Launch of goods, parts and services totaling the values;
- Component registration with serial number;
- It allows to register identifiers (brand, model, chassis, year);
- Creation general reports;
- Service orders (open, closed or all);
- Collateral management;
- Allows the launch of services based on hours;
- Access control by user and company.

5.3.4 Experimental Activity

The activity performed with subjects was aimed at verifying if documentation based on key classes could replace traditional documentation. Moreover, we investigate if the information contained in the documentation could support developers in different maintenance activities.

To perform the activity for evaluating documentation quality, we provided, for the developers, a documentation based on key classes for the target systems, which did not have traditional documentation or did not present a documentation based on key class. In this way, the developers evaluated the quality of the key classes in terms of recall and precision (presented on chapter 3) and, in addition, the quality and usefulness of the information produced from the key classes.

Regarding the activity complexity, a possible solution would be to read the content provided in the documentation to assess whether it can be a starting point to highlight relevant structural aspects of the application to guide developers during evolution activities and if it can replace or complement traditional documentation.

5.3.5 Control Questions

In this section, we present control questions to meet the understanding level and the degree of consistency in the answers of the developers. In this sense, for the questions presented below, we will control the type of response mentioned by the developers. Only answers *Disagree* or *Strongly Disagree* will be accepted considering the likert scale. In this way, for the other type of answers we will eliminate the developer of the analysis for a specific topic addressed by the question.

We created two distinct topics related to *Icons Pack* and *Banner* contained a set of questions related.

For the topic referring to *Icons Pack*. This topic contains control questions in which we expect a specific kind of reaction. This section aims to minimize the response bias, since one of the contributors was the researcher's contact.

The first control question is to comprehend if - *Icons pack are useful to comprehend architecture or design system.* Analyzing the Table 18 we can observe that a small number (on two systems) of developers disagreed. So, we did not considered answers equals to *Agree* on our result analysis for questions related to *comprehend architecture or design system*. One reason for this might be not understanding the question and having adopted a different criterion to answerer to it. The set of icons presented may indicate to the developers the abstract representation of the features and functionalities present in the systems and therefore, be useful to comprehend the design, but in particular the interaction design.

A similar situation was found in the following control sentence - *Icons pack are useful to detect smells.* Icons pack is not related for detecting smells. So, we expected that developers

Table 18 – Opinion from developers about if icons pack are useful to comprehend architecture or design system.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	66.67%	-	33.33%	-

answered (*Disagree* or *Strongly Disagree*). We can observe on the Table 19 that few developers disagreed (on two systems). We did not considered answers equals to *Agree* on our result analysis related to *detect smells*. A possible reason for this is that developers had little or no contact with the bad smells concept, and this may have affected the decision making.

Table 19 – Opinion from Developers about icons pack are useful to detect smells.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	66.67%	-	33.33%	-

A positive situation occurred for another control sentence - *Icons pack are useful to evaluate complexity metrics.* On the Table 20 the most of developers disagreed about the icons pack utility for evaluating metrics. In this situation we also did not considered answers equals to *Agree* on our result analysis for questions related to *evaluate complexity metrics*.

Table 20 – Opinion from Developers about if icons pack are useful to evaluate complexity metrics.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	66.67%	-	33.33%	-
Scholar	3	-	100%	-	-	-
Financial	3	-	100%	-	-	-

Next control sentence: *Icons pack are useful for software maintenance or introduction of new functionalities.* In this case a small number of developers also were *disagree* (Table 21). In this situation we also did not considered answers equals to *Agree* on our result analysis for questions related to *software maintenance or introduction of new functionalities*. A possible reason on icons pack is intuitive to manage features already existing on the application, so the addition, adaptation and organization to new features can be facilitated.

Table 21 – Opinion from Developers about if icons pack are useful for software maintenance or introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	66.67%	-	33.33%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	66.67%	-	33.33%	-

Next topic is another control topic named *Banner Section*. We included a company logo on the documentation based on key classes. In this topic we expected a specific kind of reaction, so this section aims to minimize the response bias. First control sentence we want to know if

Banner is useful to comprehend architecture or design system. The most of developers agreed, this was negative situation (Table 22), we can observe that a small number (on two systems) of developers disagreed. So, we did not considered answers equals to *Agree* on our result analysis for questions related to *comprehend architecture or design system* because banner is not related to comprehend architecture or design system. This was the last section of the survey that contained 36 questions. A possible explanation for the kind of answer may be related to the developer's dismay, tiredness, workload, and carelessness in evaluating that question.

Table 22 – Opinion from Developers about banner to comprehend architecture or design system.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	33.33	-	66.67%	-
Scholar	3	-	-	-	100%	-
Financial	3	-	33.33%	-	66.67%	-

Next control sentence: *Banner is useful to detect smells.* We had a positive perception of developers (Table 23). The most of developers disagreed about to use banner to detect smells. We did not considered answers equals to *Agree* on our result analysis for questions related to *detect smells*. A similar situation was verified on next control sentence *Banner are useful to evaluate complexity metrics* - (Table 24).

Table 23 – Opinion from developers about banner to detect smells.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	100%	-	-	-
Scholar	3	-	66.67%	-	33.33%	-
Financial	3	-	100%	-	-	-

Table 24 – Opinion from Developers about banner to evaluate complexity metrics.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	33.33%	-	66.67%	-
Scholar	3	-	66.67%	-	33.33%	-
Financial	3	-	100%	-	-	-

However, developers were not always coherent to answer the control sentence - *Banner is useful for software maintenance or introduction of new functionalities.* On the Table 25, we can observe that a small number (on two systems) of developers disagreed. We did not considered answers equals to *Agree* on our result analysis for questions related to *software maintenance or introduction of new functionalities*.

Table 25 – Opinion from developers if banner is useful for software maintenance or introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	33.33%	-	66.67%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	100%	-	-	-

5.3.6 Results

The results will be presented for the survey performed with 4 developers from 4 software systems (3 proprietary software and 1 open source software). We constructed a documentation based on key classes available to guide developers during comprehension activities. We created a minimum and direct documentation to contextualize key classes for developers with some useful information that helps to understand general concepts about the target software.

The research questions were answered through the questionnaire after evaluating the documentation produced around the key classes.

The survey was answered in two steps: the first step was to verify the feasibility and quality of a documentation based on key classes. The second step it was to verify feasibility and quality of a documentation based on key classes to support maintenance tasks. We applied this control on second step, so to six criteria presented in Section 5.3.1 *Documentation Quality* and *Key classes Quality* were not applied control.

5.3.6.1 Documentation Quality

We asked to the developers their opinion on the organization of key classes using dependency graphs, as an important factor for understanding the structure of the application - (*Dependency graphs are important for understanding the main dependencies of the application*). All developers answered "Agree" (Table 26). On this fact key classes could be used to enable developers focus on the design of the target system.

An aspect we want to investigate is the possibility of establishing a design view using a dependency graph on key classes, and whether this structure is able to reveal important dependencies of source code such as circular dependencies as showed in Figure 16 (*PDDocument* ↔ *COSWriter* and *PDDocument* ↔ *PDFParser*) extracted from PDFBox using key classes, which are considered problematic. More specifically, a dependency graph of key classes can provide a meaningful view of the design, so we propose to create a dependency graph of key classes because it could reveal inconsistencies or provide additional information. In case in that the documentation is available, but it does not necessarily match what is in the source code, either because it shows only a simplified picture, or because it is outdated. The dependency graph of the main classes of the system can be useful source of information to assess design because they show the implicit dependencies and display undesirable dependencies such as circular dependencies when they exist.

Table 26 – Importance Level for Dependency Graphs.

Software	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	-	-	100%	-
Finanial	3	-	-	-	100%	-
PDFBox	1	-	-	-	100%	-

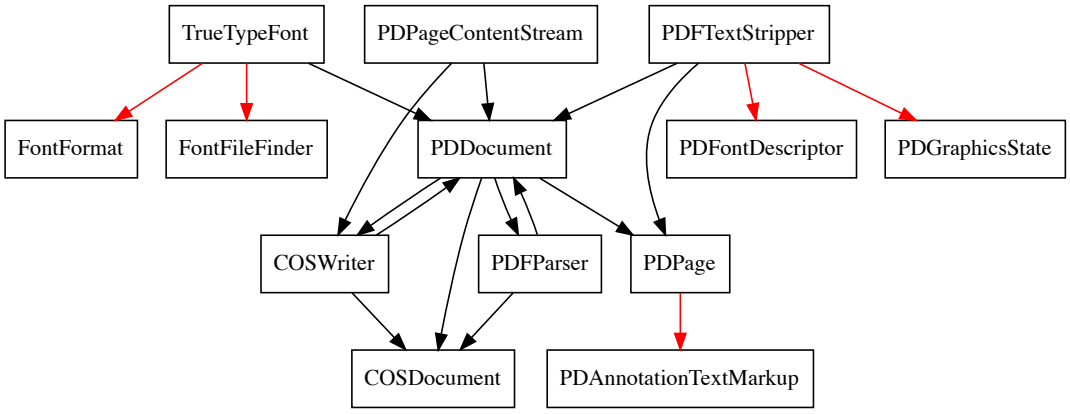


Figure 16 – Dependency Graph for Apache PDFBox.

The sentence to investigate: *Key classes are important entities for design/architecture comprehension.* In fact, for all systems their developers agreed (Table 27) with that affirmative because those classes have an important role on the system, and there are classes (considering key classes) that are more important than others, and have more impact in software design. In addition, key classes tending to present more structural anomalies compared to other system classes, this can be linked to the strong control that those classes have on the application. Thus, this finding suggests that focusing the study on key classes would help developers to assess the overall design and possibly indicate the critical parts of the system that need attention in order to improve that overall design quality.

Table 27 – Importance Level of Key Classes for Design/Architecture Comprehension.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	-	-	100%	-
Financial	3	-	-	-	100%	-
PDFBox	1	-	-		100%	-

Next sentences intend to evaluate the quality of the presented documentation. So, we asked if - *This new documentation is a design documentation.* The PDFBox developer do not have an opinion. Although, this result is apparently negative, it was somehow expected. One of the reasons is that the PDFBox application has a traditional documentation available. Another factor relates to the developer's experience level with strong knowledge of the code in some cases eliminates the need for documentation. On the other hand, in environments where there is no documentation available, the reaction of the developers was positive in most cases (see Tables 28 e 29). In this situation, we observe mainly that a documentation based on key classes can complement the traditional documentation of the original developer used in application.

In sequence, we asked if - *The provided information is useful to complement the understanding of the general organization of the system.* The most of developers agreed about this

Table 28 – Opinion from Developers about Design Documentation.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	-	-	100%	-
Financial	3	-	-	-	100%	-
PDFBox	1	-	-	100%	-	-

affirmation (Table 29). The set of key classes highlights classes that are important from the design viewpoint, since they present important structural properties, and therefore are able to represent a general organization of the system.

Table 29 – Opinion from Developers about if the provided information is useful to complement the understanding of the general organization of the system.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	-	-	66.67%	33.33%
Financial	3	-	-	-	100%	-
PDFBox	1	-	100%	-	-	-

We asked if - *This new documentation can replace the traditional documentation of the original developer used in application.* On Table 30 developers do not have a clear opinion about this topic. A reason about this could be the documentation based on key classes is produced using dynamic analysis, providing a straight relation to the actual behavior of the software would benefit cognitive activities, and therefore producing more accurate solutions during comprehension activities. However, key classes could not cover all details necessary for understanding the systems, and thus additional information would still be necessary.

Table 30 – Opinion from Developers about if the new documentation can replace the traditional documentation of the original developer used in application.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	67.67%	33.33%	-
Scholar	3	-	-	33.33%	66.67%	-
Financial	3	-	-	100%	-	-
PDFBox	1	-	100%	-	-	-

We want to evaluate whether a general documentation in key classes is easy to understand - *The new documentation is easy to understand.* Table 31 showed that the most of developers had positive perception level in relation for comprehension facility (property systems developers). One possible explanation for this is due to the absence of reference documentation or a criteria for producing documentation that could be used as a benchmark. The PDFBox developer *disagree*. Although, this result is apparently negative, it was somehow expected. One of the reasons is that the PDFBox application has a traditional documentation available. Another factor relates to the developer's experience level with strong knowledge of the code in some cases eliminates the need for documentation.

Next sentences are regarding key classes information quality. First, PDFBox developer points out that there is missing information in the generated documentation - *There is missing information in the generated documentation*. Analyzing Table 32 we credit this problem to

Table 31 – Opinion from developers about comprehension facility.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	-	-	100%	-
Financial	3	-	-	100%	-	-
PDFBox	1	-	100%	-	-	-

the reasonably low recall. However, we need to mention that the set of mined key classes is not definitive, mainly because of two points. First, the listing of key classes is sensible to target number of key classes k defined by the user. If the user wants to manage more detail, a higher k is selected. On the other hand, a lower k may be selected to assess just a few key classes. The other point is that the set of key classes may vary between different versions of the system. In this study, we applied the approach on the last version of the target systems, so the set of key classes may not be same compared to other versions because of design evolution. To analyze design evolution based on key classes, different values for k should be analyzed based on the assumption that as the size of systems tend to grow so would be the set of key classes.

On the other hand, in environments where there is no documentation available, the reaction of the developers was neutral for all cases. We believe this reaction is due to absence of a standard documentation to compare the available information making it a barrier to disseminating architectural knowledge.

Table 32 – Opinion from Developers about missed information in the generated documentation.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	100%	-	-
Scholar	3	-	-	100%	-	-
Financial	3	-	-	100%	-	-
PDFBox	1	-	-	-	100%	-

For next research questions were answered only by proprietary system developers and applied the control questions to clear the answers. We contacted PDFBox developer to answer some more questions related to the effectiveness of the documentation for different tasks, but we did not have feedback.

5.3.6.2 Quality of Key Classes

The next sentence is intended to know if - *Key classes set are adequate starting point to comprehend the application.* Analyzing Table 33 we can conclude a positive perception in relation to the starting point to comprehend an application. Documentation based on key classes is simple and straightforward, because the set of key classes tends to be small. Therefore, documentation based on key classes would have help on decreasing system understanding time. The rationale is that a small set of key classes can guide the developer more quickly rather than navigating on all available source files, in case when documentation is not available. On the other hand, key classes would be more complex than ordinary classes, and still understanding would not be simple.

Table 33 – Opinion from Developers about key classes set are adequate starting point to comprehend the application.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	33.33%	-	66.76%	-
Financial	3	-	-	-	100%	-

Other sentences are intended to know if:

- *Knowing key classes are useful information for software maintenance.* - Table 34;
- *Knowing key classes are useful information for introduction of new functionalities.* - Table 35;
- *Knowing key classes are useful information for bug fixing.* - Table 36;
- *Consider complexity metrics boxplots and bad smells information. If your system has some architectural problems could be solved restructuring key classes.* - Table 37.

For these sentences the most of developers agreed. This reaction is expected for us. Documentation based on key classes highlights structural properties important of the application and therefore they are a guide in maintenance distinct activities.

Table 34 – Opinion from Developers about if Knowing key classes are useful information for software maintenance.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

Table 35 – Opinion from Developers about if key classes are useful information for introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

Table 36 – Opinion from Developers about if knowing key classes are useful information for bug fixing.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	-	-	100%	-
Scholar	3	-	66.67%	-	33.33%	-
Financial	3	-	-	-	100%	-

5.3.6.3 Smell Detection Utility

We formulated *Smell Detection Section* to evaluate some sentences such as if - *Detected smells presented in the documentation are useful to show design anomalies.* The developers agreed with this sentence as shown on the Table 38. As shown in an earlier chapter some bad smells

Table 37 – Opinion from Developers about consider complexity metrics boxplots and bad smells information.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

Table 38 – Opinion from Developers about if smells presented in the documentation are useful to show design anomalies.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

such as long parameter lists and complex classes are related to high coupling, so these are a natural way to improve these modularity indicators.

A similar and positive perception (Table 39) was observed on the next sentence to know if - *Knowing detected smells are useful information for software maintenance.* This is expected for us, because as most bad smells are concentrated on key classes, then developers should prioritize them due to their higher impact on the overall design.

Table 39 – Opinion from Developers about if knowing detected smells are useful information for software maintenance.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

We asked if - *Knowing detected smells are useful information for introduction of new functionalities.* One developer did not have concrete opinion about this (Table 40). A possible reason would be that developers had not yet experienced a similar situation, for example using bad smells information for introduction of new functionalities.

Table 40 – Opinion from Developers about if knowing detected smells are useful information for introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	100%	-	-	-
Financial	2	-	-	-	100%	-

However, a positive perception was collected for the sentence - *Knowing detected smells are useful information for bug fixing.* A possible reason is the fact bugs can be related to design anomalies and consequently to bad smells as shown on the Table 41.

5.3.6.4 Trace Tree Utility

We also created a section for evaluating the documentation based on key classes regarding trace tree utility. The first question is if - *Trace trees presented in the documentation are useful information to show design anomalies.* The most of developers agreed with this sentence (Table

Table 41 – Opinion from Developers about if knowing detected smells are useful information for bug fixing.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

42). In fact, trace trees can show unwanted call sequences and consequently can be useful for evaluating design anomalies. So, developers can benefit from tree structure and recommend changes during maintenance activities as shown on Table 43 for evaluating the following sentence: *Knowing trace trees are useful information for software maintenance.*

Table 42 – Opinion from Developers about if Trace trees presented in the documentation are useful information to show design anomalies.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	33.33%	-	66.67%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	33.33%	-	66.67%	-

Table 43 – Opinion from Developers about knowing trace trees are useful information for software maintenance.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

In sequence, we asked if - *Knowing trace trees are useful information for new functionalities.* Trace trees are useful for feature location and the developers (Table 44) can use their structure to find a adequate strategy to include new functionalities minimizing the degradation of the system structure.

Table 44 – Opinion from Developers about if knowing trace trees are useful information for new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	50%	-	50%	-

The developers in general presented positives (Table 45) to use trace trees information for bug fixing - *Knowing trace trees are useful information for bug fixing.* A bug may be associated with a particular feature of the application. Developers can use the information available in the tree structure to evaluate sequence of called classes and methods to find bugs.

Next sentence we are going to know if - *Graphs "All usages" presented in the documentation are useful to show design anomalies.* The most of developers as show on the Table 46 agreed. Graphs "All usages" show all classes used by a specific key class and all classes which use a specific key class. So, this graph can be useful to show undesirable anomalies based on relationship among classes.

Table 45 – Opinion from Developers about if knowing trace trees are useful information for bug fixing.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	66.67%	-	33.33%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	-	-	100%	-

Table 46 – Opinion from Developers about Graphs "All usages" o show design anomalies.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	33.33%	-	66.67%	-
Scholar	3	-	33.33%	-	66.67%	-
Financial	3	-	-	-	100%	-

The following sentence - *Methods and attributes presented in the documentation are useful for software maintenance or introduction of new functionalities.* In general developers agreed (Table 47). Methods associated to roots of the key classes in the execution trace tree have been highlighted, for example showing code examples and their role in the class. In addition, other important methods contained on a key class have also been emphasized. As key classes have a high impact on software design, evaluating information on such methods can be useful in assessing the impact of the new feature.

Table 47 – Opinion from Developers about if methods and attributes presented in the documentation are useful for software maintenance or introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	100%	-	-	-
Financial	2	-	-	-	100%	-

5.3.6.5 Dependency Graph Utility

Dependency Graph Section is intended to evaluate the opinion of the developers if - *Knowing dependency graph are useful information for bug fixing.* As shown on the Table 48 developers considered useful to use dependency graph for bug fixing. In fact, key classes dependency graph can provide a starting point to investigate and locate classes that contain bugs. Because bugs can be associated to undesirable dependencies mainly to key classes which have high impact on the design software.

Table 48 – Opinion from developers about if knowing dependency graph are useful information for bug fixing.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	3	-	66.67%	-	33.33%	-
Scholar	3	-	-	-	100%	-
Financial	3	-	-	-	100%	-

Developers had a positive perception about to use dependency graph for software maintenance and introduction of new functionalities: *Knowing dependency graph are useful information for software maintenance* - (Table 49) and *Knowing dependency graph are useful information for introduction of new functionalities* - (Table 50). In fact, dependency graph show an

overview of the most important relationship of the application and so, this an relevant information for evaluating the impact to include new functionalities during software maintenance activities.

Table 49 – Opinion from Developers about dependency graph for software maintenance.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

Table 50 – Opinion from Developers about if knowing dependency graph are useful information for introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	-	-	100%	-

On the sentence - *Knowing dependency graph are useful information for detecting smells.* Developers agreed (Table 51). A possible reason is that dependency graph presents an overview of the key classes in terms of relationship. If classes are more prone to present high coupling and low cohesion, so they can be a adequate starting point to investigate bad smells information.

Table 51 – Opinion from developers about if Knowing dependency graph are useful information for detecting smells.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Scholar	1	-	-	-	100%	-
Financial	2	-	50%	-	50%	-

5.3.6.6 Complexity Metrics Utility

On section named *Complexity Metrics*, firstly we asked to developers if - *Considering complexity metrics boxplots. Boxplots set are enough for evaluating architectural problems of the application*. This result (Table 52) is expected for us, complexity metrics are adequate to analyze architectural problems, because they can evaluate cohesion and coupling measures.

Table 52 – Opinion from developers about if boxplots set are enough for evaluating architectural problems of the application.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	2	-	-	-	100%	-
Financial	3	-	-	-	100%	-

Next sentence is intended to verify if - *Knowing complexity metrics are useful information for introduction of new functionalities.* The most of developers had negative perception about this context (Table 53). One possible reason for this may be related to the knowledge high level about the code since these developers are the owners of the application and therefore do not

feel the need to evaluate of the application complexity as an initial step to the addition of new functionalities. On the Tables 54 and 55 we asked the respective sentences: *Knowing complexity metrics are useful information for software maintenance;* and *Knowing complexity metrics are useful information for bug fixing*, developers were undecided. We believe that this is due to the lack of experience with this kind of evaluation during maintenance activities.

Table 53 – Opinion from Developers about complexity metrics are useful information for introduction of new functionalities.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	100%	-	-	-
Scholar	1	-	-	100%	-	-
Financial	2	-	100%	-	-	-

Table 54 – Opinion from developers about complexity metrics for software maintenance.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	1	-	-	-	100%	-
Financial	2	-	50%	-	50%	-

Table 55 – Opinion from developers about complexity metrics for bug fixing.

Software/Likert scale	# Developers	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Service Order	1	-	-	-	100%	-
Scholar	2	-	50%	-	50%	-
Financial	3	-	66.67%	-	33.33%	-

5.4 Threats to Validity

There are several threats to validity that we will categorize in relation to subjects (students and developers), activities and others.

5.4.1 Subjects

Participant's willingness to participate: the participant's performance during the execution of an activity is influenced by the subject at that time available. The subject may mood be diverse: excited, tired and upset and this factor may interfere with his concentration and comprehension capacity. In experiments with students groups in institutions were performed on the same day and, it was just one comprehension activity. For developers, the documentation assessment based on key classes was performed in day, place and schedule of their choice. Even the delivery date of the questionnaires was also negotiated, when they could not meet the deadline. Another factor that mitigate this threat is that the participants were volunteers and they were told they could give up the activities at any time.

Familiarity of the participants with the documentation: the environment and the lack of clarity and knowledge of the technical terms used in the documentation may interfere on the results. In addition, there may be a lack of clarity about the objectives and details of carrying

out the activities. To mitigate this threat, to the students groups instructions were provided and they had contact with the environment used. However, learning and adaptation can vary among subjects and therefore interfere with results. About the developers, the same threat is observed for developers and moreover, it is necessary to consider the workload and the venue to assess the documentation chosen by himself.

Subjectivity of the definition about which elements of the documentation and code are useful, complex or important to the execution of the activities from the point of view of the participant. Since each element of the documentation can have a distinct relevance level for each subject, according to the knowledge and experience level of the participants. To mitigate this threat we consider an adequate number of participants in the experiments. Another threat is related to the criteria based on Likert scale has individual interpretation. To mitigate this threat it is necessary to well define the meaning of each value in the scale to avoid bias interpretation.

Subjects may have some kind of knowledge about the development new approach. To mitigate this threat, no subject involved with the research participated in the experiments. The subjects did not know the questions and did not know what documentation was the result of the approach. The developers knew they were evaluating the documentation produced by our approach. To mitigate this threat, we consider some control questions to minimize the influence of the positiveness of the subjects.

5.4.2 Activities

Complexity of tasks. They can impact on the time. To mitigate this threat, simple activities were proposed. A situation is observed on the developers and students is about the difficulty degree is related to the activity complexity (required knowledge), such as classes, methods, documentation analysis that employ the use of terms that are unknown among the subjects involved. Another way to mitigate this threat was to instruct students and developers presenting the organization of documentations and concepts that would not usual among participants.

Activities may have been difficult for the experiment. However, most of the participants completed the activities within the time limit and there were no withdrawals. Another factor taking into consideration was the low student experience and the duration of the experiments. In this way, the planning and the complexity of the activities took into account these two factors. Within experiment with developers the time was not estimated to complete to analyze the documentation and the level of the activities was simple, minimizing effect of the threat.

5.4.3 Other Threats

The size and content of the documentations. can influence the ability to query and use them. The two types of documentation (traditional and based on key classes) presented distinct content, but similar in relation to the size. Thus, the effort to evaluate the two documentation would

not be influenced by the size of the documentation.

Experimenter Effects (researcher). This effect may occur because the researcher is involved with the proposal. In this could, there may be unconscious influence of the researcher when instructing the participant subjects, choosing questions, selecting the experiments, etc. In the adhesion document of the participant, the subjects were asked for strict sincerity in their answers.

5.4.4 External Validity

The generalization of our results can be hampered by the limited representativeness of subjects and tasks.

Ideally, subjects sample should represent all possible users of the approach in a real scenario. The sample is varied, as it considered subjects of academia and software industry. However, most of the subjects are students who do not have in-depth experience with development. We contacted developers of open source and proprietary systems, of which only 4 participations were collected. Thus, subjects sample does not represent the profile of possible end users for this approach. The activities also do not cover all potential uses of key class information. To cover these possibilities, the number of necessary activities would be incompatible with the participants' availability.

5.4.5 Construction Validity

The first contact with the application can influence the second contact: to minimize this influence the groups used only one kind of documentation. However, this can not be fully controlled, for instance, the developer had contact with the code files and who had contact with the documentation can become familiar with the organization of the application.

5.5 Discussion

To better understand the achieved results, we will consider the responses for the open questions to help identify problems, needs and opinions of subjects.

5.5.1 Experiment with Students

This experimental setup involved the design comprehension using the available documentation of the systems as support material. Two institutions agreed to collaborate with our experiments. Our aim was to investigate if a documentation based on key classes can complement or replace the traditional documentation, and whether it is effective for software maintenance tasks, such as, application design comprehension. Considering the evaluated criteria, in general, we can

affirm that no significant statistical differences were found between the evaluated documentations. In particular, this result is positive to our approach, indicating that the approach for mining key classes and the information retrieved around them, can be used as a starting point for understanding the application during evolution activities and it can potentially replace or at least, complement traditional documentation.

We used thematic analysis (CRUZES; DYBA, 2011) to produce a systematic outcome of the students responses in open questions. The researcher read all responses, and labeled responses with open codes that specified the realm of the response. Responses with similar realms would be labeled with the same code. After all responses were labeled, the researcher verified related codes within a same theme. Then, the final themes were reviewed and named.

We present themes that emerged after analyzing the answers received from the questionnaire. We present a discussion considering four open questions and give some examples of answers associated to them.

The first open question is about usefulness.

Q1- About the information presented in the documentation, how was it useful to comprehend the application? Table 56 summarizes the results for *Q1*.

Table 56 – Themes that emerged after analyzing the answers received from the questionnaire - Question Q1.

Theme	# Occurrences on the survey	Institution	Key Classes (#occurrences)	Key classes + Traditional (#occurrences)	Traditional (#occurrences)
Non-specific	14	1, 2	-	8	6
Application overview	12	1, 2	8	-	4
Description of classes	9	1, 2	6	3	
Versions	6	1, 2	-	2	4
Visual representation	6	2	4	-	2
Command Line	3	1, 2	-	1	2
Coding conventions	2	1	-	-	2
Contact of developers	1	1	1	-	-

We also asked to the students how they conducted the analysis of the documentation:

Q2- Describe how you performed the comprehension provided documentation (how you read and analyzed the documentation). Table 57 summarizes the results for *Q2*.

Table 57 – Themes that emerged after analyzing the answers received from the questionnaire - Question Q2.

Theme	# Occurrences on the survey	Institution	Key Classes (#occurrences)	Key classes + Traditional (#occurrences)	Traditional (#occurrences)
Selection of topics	15	1, 2	7	4	4
Visual representation	12	1, 2	7	5	-
Reading based on title	7	1,2	-	-	7
Based on functionalities	6	1	-	6	-
Detailed reading	6	1	4	-	2
Non-specific	4	1	-	4	-

The third question is aimed at finding the points influenced his/her performance.

Q3- What factors do you attribute to your success/failure in performing the activity? Table 58 summarizes the results for Q3.

Table 58 – Themes that emerged after analyzing the answers received from the questionnaire - Question Q3.

Theme	# Occurrences on the survey	Institution	Key Classes (#occurrences)	Key classes + Traditional (#occurrences)	Traditional (#occurrences)
Non-specific	9	1, 2	-	4	5
Application overview	9	1, 2	-	9	-
Versions	9	1, 2	-	5	4
Description of classes	8	1, 2	5	3	-
Documentation organized	6	1, 2	-	6	-
Visual representation	4	2	3	-	1
Idiom	4	1, 2	1	1	2
Lack of experience	4	1, 2	2	1	1

Finally, we asked to the students:

Q4- Describe the information that attracted you more and that helped you in describing the design description. Table 59 summarizes the results for Q4.

Table 59 – Themes that emerged after analyzing the answers received from the questionnaire - Question Q4.

Theme	# Occurrences on the survey	Institution	Key Classes (#occurrences)	Key classes + Traditional (#occurrences)	Traditional (#occurrences)
Code	15	2	6	5	3
Non-specific	10	1, 2	4	3	3
Visual representation	8	2	-		8
Organized documentation	7	2	-	7	-
Application overview	5	1	1	3	1
No information, because my lack of knowledge	3	1, 2	1	1	1
Software operation	2	2	-	-	2
Idiom	2	1, 2	1	1	-
Versions	1	2	-	-	1
Smells	1	2	1	-	-
Command line	1	2	-	-	1

Regarding the themes emerged among open questions, we are going to present a possible explanation.

Application overview: Among the answers analyzed emerged from the questions *Q1, Q3* and *Q4*, the most relevant information pointed by the students regarding the usefulness is the *application overview*. We have, as example: *The vital information for the understanding of the project was those available in the overview and migration/getting started part, because it presented the project and show how to work with the project, respectively.* One possible explanation is related to time. Regarding questions *Q3* and *Q4* key classes + Traditional group, this theme was useful to comprehend the application, in particular this group evaluated two documentations under limited time, so they adopted a superficial analysis of the documentation.

Selection of Topics: Among the answers analyzed emerged from the question *Q2*, the most prevalent way of performing the comprehension was *selecting topics* to deepen the analysis. We have, as examples: a)*By random topics* and b) *I went through the pages and reading what*

I found pertinent. I searched for words that excited my curiosity or that I found interesting. We believe this situation occurred because comprehension activity was limited to a maximum time of 60 minutes and consequently the students have also chosen to prioritize some topics. The selection of topics had more occurrence in key classes group, one possible explanation is related to documentation structure based on key classes. For each key class, the student found the same correspondent content, so they decided filter some classes to visit.

Reading based on the title: This topic emerged from the question *Q2*. This kind of reading was generated due to the short time that the students had to complete the task, being guided by the matter described in the titles of the documentation that catched the interest of the newcomer. In this context, one important aspect to be mentioned is the clarity and objectivity of the information. We have, as example: *The information given in the titles and anchors, and pages giving a "summary" of the theme.* The reading based on title had more occurrence in Traditional group, a possible explanation is related to volume of content (item) on the documentation that naturally induced the students to perform a reading based on titles attractive.

Reading based on the functionality: This topic emerged from the question *Q2*. This kind of reading also was generated due to the short time that the students had to complete the task. Another possible explanation is related in the application comprehension by identifying the most important features of the application. We have, as example: *The application's features are all separated and grouped, making it easy to read the documentation.* The reading based on functionality had more occurrence in Key classes + Traditional group, a possible explanation is related to comprehend the application design considering the features of interest.

Detailed reading: This topic emerged from the question *Q2* was observed to groups of students who performed the analysis of only documentation and decided to get into the details of the application design. We have, as example: *I read the documentation in order to abstract as much detail as possible. It was possible to understand everything about the application and how to use it...* This theme occurred on the key classes group, because the documentation was simple and direct centered on software design.

For the above themes, they are related to the time to perform the comprehension task. In both institutions regarding the time, there was no significant statistical difference among the times. But, by the qualitative analysis of boxplots, the *Key classes* group concluded the analysis in a slightly shorter time compared to the other groups. One possible explanation is that the documentation based on key classes is simple and direct and involve many visual elements such as graphs and tables, thus documentation analysis would be faster.

So, in real-world development environments, because time is a crucial element in software activities, it is important to consider an effective strategy that supports developers during the software comprehension activities. Thus, a documentation containing design information, such as structural aspects, is relevant to developers to comprehend important aspects of an application.

Lack of experience: This topic gathered answers from students to question *Q3* related to the difficulty they encountered in understanding the software design. One possible explanation is related to the students' lack of contact with design documentation, the size and complexity of the evaluated application and the lack of professional experience. We have, as example: *lack of knowledge related to design documentation analysis.* This theme occurred with more frequency on the key classes group, because the students do not have professional experience with large projects and contact with the documentation centered on design specific for developers comprehend a design overview.

Idiom used in the documentation: Among the answers analyzed from question *Q3* and *Q4*, the information pointed by the students is a barrier related to the idiom used in the documentation. As example, we have: *documentation was in English, which made it difficult to understand.* For question *Q3* this comment was frequent to *Traditional* group and for question *Q4* occurred a regular distribution between Key classes and Key classes + Traditional group. A possible explanation for question *Q3* is that traditional documentation has more textual descriptions while documentation based on key classes emphasizes visual representation of the information.

Organized documentation: Among the answers analyzed from question *Q3* and *Q4*, the most relevant information pointed by the students is related to organization of the documentation. We have, as example: *The documentation is very explanatory, what to do and what not to do, and all the functionalities are shown and exemplified in a page responsible for describing some algorithms and show the features of the library.* This theme was observed on key classes + Traditional group, a possible explanation was available content in structure adequate that facilitates the navigation.

Visual representation: Among the answers analyzed from questions *Q1, Q3* and *Q4*, the most relevant information pointed by the students is related to visual representation, mainly dependency graphs available in documentation based on key classes. We have, as example: *The information that attracted me most was the general description of the classes using the dependency graph, because in it is possible to understand the application and which specific place I should go to reach a certain functionality.* For question *Q3* visual representation them is related to dependency graphs reported by *Key class* group, while at the Traditional group reported considering the available screenshots.

This theme is related to the *Usefulness Information* criterion, there were no statistical significant differences. However, group *Key classes + Traditional* of institution 1 had a positive perception about the useful information. We believe that this result is related to the contact with a larger amount of information by this group.

Code example: Among the answers analyzed from question *Q4*, the most relevant information pointed by the students is related to visual representation. We have, as example:*The part that called attention was to show users code examples, since most of the time the users (mainly*

programmers without so much experience) take a documentation does the reading but ends up not understanding how to apply that to their project, then giving examples of how to apply the library to the project makes understanding easier. this theme occurred in a regular distribution among all groups.

This theme is related to *Methods utility* criterion. No significant statistical differences were found between groups. In particular, Key classes + Traditional group of institution 1 pointed out that the highlighted methods highlighted. In both documents, methods details from distinct features were presented, however preference over traditional documentation was due to the presentation of complete examples of the features which increased the application comprehension perception, while in the documentation based on key classes only the role of key classes methods was evaluated.

Description of classes: This theme emerged from answers to question *Q1* and *Q3*. Both documentation contained references to the description of classes, but in different ways, and often mentioned by students. In addition to code example, the documentation contains textual descriptions that present the classes overview. A possible explanation for this, is that code example connected to the textual description of the classes, enables the student to quickly learn about the features of interest. We observed a regular distribution of this theme among the groups, a possible explanation is that textual descriptions guide the newcomers to comprehend the code faster.

Versions: In this topic refers answers emerged from question *Q1, Q3* and *Q4* regarding to traditional documentation content, it focuses on downloading available code versions from version control software such as SVN, GitHub. This interest was already expected by us, since developers need to resort to these repositories when they decide to make use of the target application in a new project. We observed a uniform distribution of this theme among the group which evaluated traditional documentation. We have, as examples: *Links of the repositories: GitHub, Svn, facilitates the access to the files of way and practice. Download Links: Download the application, easy visibility and download the application.*

Command line: In this topic emerged from question *Q1* and *Q4*, also refers to traditional documentation content from PDFBox. Command line comes with a series of command-line utilities. A possible explanation is reported from a student: *it was the main one to understand what the application is capable of, and how to use it.* We observed a uniform distribution of this theme among the groups which evaluated traditional documentation.

Coding Convention: In this topic emerged from question *Q1*, also refers to traditional documentation content from PDFBox. This interest was already expected by us, because it is specif for development. We have, as example: *it is useful about formatting, white space, comments, variables, etc.*

Contact of developers: This topic emerged from question *Q1*, refers to documentation

based on key classes content. It was mentioned from one student from Key classes group who reported: *and if there is still doubt, it is possible to contact the developers.* In particular, this topic can assist newcomers in environments where documenting is not available, the code is difficult to understand, etc. A possible explanation for the lack of interest in this topic is the lack of professional experience of the students.

Smells: One student belonging Key classes group reported on this topic on question *Q4*, regarding the way as code smell was represented in the documentation based on key classes as reported: *I was struck by how smells detection was demonstrated*. A possible explanation for the low interest of students in the subject is the novelty of the subject, since the topic is not completely covered in the courses.

Software operation: This topic emerged from question *Q4*, gathered answers related to how to use the software, how to access the functionalities of the software. Two students from traditional group reported about this theme. We have, as example: *I tried to read in a quick way, trying to absorb what the software did, what the main features.* In particular of this theme is specific from traditional documentation available.

Non-specific: This theme gathers answers where it was not possible to identify the intention of the participant. One possible explanation is the discouragement, tiredness, or lack of understanding of the purpose of the question. We have, as examples: a) *Success*, b)*Focus on task execution.* c) *I know English and I like extracurricular readings.* In general we observed a regular distribution this theme among the groups for all questions. In particular, for question *Q2* this theme occurred only to Key classes + Traditional group.

No information, because my lack of knowledge: This topic emerged from question *Q4* on the gathered answers. In this case the students evaluated the documentation, but did not assimilate knowledge related to software design. A possible explanation is related technical terms unfamiliar to students and software complexity in relation to the size.

From the Tables 56, 57, 58 and 59 the themes with many answers are: visual representation, organized documentation, overview, code, description of classes and selection of topics. In particular, these themes reflect the basic needs of newcomers who decide to engage in a new project and need to understand and assess the design during maintenance tasks. In relation to other themes with less occurrence of answers are specific to the target software knowledge domain and therefore, less usual in documentations.

5.5.2 Survey with Developers

On the survey with developers we establish some relevant perceptions about the quality of the documentation produced around the key classes, after analyzing the questions that were measured with Likert scale. Some perceptions of the developers were not consistent with the expected, but in general, the analysis of the questionnaires allowed us to collect the perceptions

described in sequence.

Documentation centered on design: Regarding to the **usefulness** of the documentation based on key classes the developers presented a positive perspective, this can be a result of the reasonable recall and precision obtained. Another positive aspect from the developers was to suggest that key classes and other resources (trace trees, dependency graph, etc.) are an important starting point to comprehend the application design. We conclude that key classes can to produce relevant information to guide developers in maintenance activities, introduction of new features, bug fixes, complexity metrics and bad smells information and can solve application structural problems, such as detect anomalies in the project.

Documentation based on key classes complements original documentation: Another important information observed was the positive opinion of the developers about the possibility of the documentation based on key classes to complement original documentation - (**documentation satisfaction**) criterion because it organizes the structure of the system using diagram instead on only textual descriptions long.

Experience level: software documentation may be subject to different analyzes that vary with the developer's knowledge and experience on the application. In this context on the **learning obstacle criterion** this can be sensitive to the way and the kind of information presented in a documentation. Developers who were asked about this aspect did not present an opinion on this criterion. But they have partially indicated not to be easy to comprehend the documentation (**ease for understanding criterion**), since it brings a concept-centered approach to design assessment.

Lack of documentation: Developers surveyed are the owners of the systems and they used to perform activities related to software maintenance without resorting to the project documentation. This suggests that preventive measures indicated in documentation, which ensure the structural quality of the software may be compromised. In addition, for newcomers, lack of documentation is a factor limiting understanding of the application.

Lack of knowledge based on technical terms: Based on the control questions, we observed the difficulty of some developers in judging the usefulness of the documentation regarding to technical terms such as smells, metrics of complexity, etc. Although developers were instructed in defining these terms, but the practice in performing maintenance activities on a daily basis, without considering structural aspects of the project recorded in documentation based on key classes, may compromise the maintainability of the system.

5.6 Concluding Remarks

This study provided an investigation about the usefulness of key classes organized on a structured documentation. The collected observations have implications on the field of software

architecture.

Understanding software structures, through prioritization of the key classes, which have a strong impact on the other classes of the system, seems to alleviate understanding barriers faced by newcomers. When we add information organized around the key classes, we get a simplified documentation that can be used as a starting point to understand the internals of a software.

In particular, for senior developers a documentation based on key classes enables warning of potential problems, such as low cohesion and high coupling, circular dependencies that violate a software structure, and code smells information that limits system flexibility in evolution tasks.

A fundamental aspect about the usefulness of key classes is how they are presented. In this case, we collected some information on our observations with respect to documentation:

- Documentation containing snippets of code is a cognitive barrier to newcomers. The reports collected in our questionnaire show dissatisfaction when this resource is used, unless it comes accompanied by a brief explanation.
- Long documentation, accompanied by many textual descriptions, hinders newcomers' orientation. In some cases, the newcomer decides to prioritize some topics to conduct documentation review.
- Documentation containing graphical views that reflect the organization of system internals, facilitates understanding the application, and thus activity are likely to be completed in less time.
- Finally, the absence of widely used criteria to describe the design of an application makes it difficult to analyze the quality of documentation around the key classes.

In the next chapter, we will present studies that address topics, ranging from architecture recovery techniques to structural and social problems in software.

CHAPTER 6

Related Work

This section presents studies that address topics, ranging from architecture recovery techniques to structural and social problems in software. Studies that deal with the problems of program comprehension are also important since these highlight the main difficulties encountered, when dealing with the large amount of data that is available to developers and their need to understand. Keecle in turn retrieves a reduced data set as a starting point for understanding important software structure and relationships. Along the same line, studies dedicated for data visualization are presented.

6.1 Program Comprehension

Research on program comprehension has already developed several strategies for presenting data in different areas as shown below. In general, we observed that most of the reviewed papers in this section show that when considering static and dynamic data analysis, data reduction is greater for static data in most cases. When considering the reduction of the size of the traces, the authors do not generally discuss the used procedure. Another aspect concerns limitations of some techniques to support the analysis of large trace files, i.e, only a limited number of studies suggest solutions to a few classes or method applications. The authors concluded reporting three lessons learned: First, they observed that the feature location activity sets an example in the way research results are evaluated. Second, the standard object-oriented systems may be overemphasized in the literature at the cost of Web applications, distributed software, and multithreaded systems, for which the authors have argued that dynamic analysis is very suitable. Third, with regard to evaluation, the comparisons and benchmarking do not occur as often as they should, particularly in activities other than feature location.

Problems on program comprehension have motivated the emergence of numerous approaches of dynamic analysis with different techniques and tools. In a previous work Cornelissen et al. (2009) presented a systematic review to contextualize and investigate this set of proposals, based on dynamic analysis, providing an overview of the main contributions of the field, serving as

a source of support for identifying gaps and opportunities. The authors analyzed 176 selected articles. Articles were classified taking into account four criteria: activity, target, method and evaluation.

The study by Trumper, Dollner and Telea (2013) investigated the understanding of program execution through traces. An important task in this context is the comparison of two traces for finding similarities and differences in code execution, execution order and execution duration. For large and complex traces, the difficulty related to the cardinality (size) of the trace data was tackled with a new visualization method based on packages. The technique aimed at helping users to navigate the main differences between two traces by using the TRACEDIFF tool. However, a factor which affects the tool quality is the difficult of using and understanding the information shown in the tool, which causes misunderstandings.

The work of Dugerdil (2007) proposed reduction technique for execution traces based on a sampling strategy. The concept of an omnipresent class is defined as an analogical signal processing for noise. During analysis, omnipresent classes may be removed to focus the analysis only on the relevant classes. Using the technique of samples, correlates elements and dynamically checks that they occur in the same samples. In this manner, classes can be dynamically grouped for recovering components in already existing systems. In the case study, the technique is applied in a relatively complex system with 240 KLOC on the client and 90KLOC on the server.

The work presented by Sartipi and Dezhkam (2007) suggests the use of dynamic and static views of a system software. The representation of a dynamic view was defined by collecting useful information from the execution of a set of scenarios that covers the features. The information obtained is embedded in a process of static view recovery. The approach combines static and dynamic information in a architectural recovery technique based on patterns.

In the work of Briand, Labiche and Leduc (2006) a methodology was proposed to recover of sequence diagrams using the dynamic analysis on distributed systems in Java. In this approach, while collecting traces, loops and recursive calls are not inserted. The tool to collect traces is based on AspectJ, similar to that used in this study, which captures the name of the qualified method, the call level in the execution stack, timestamp, identifier of the object, and also collects the parameters and retrieves return values of calls methods, control flows as if, else, while, for, do, switch case and return. A limitation of this work is the difficult understanding of large sequence diagrams, due to the amount collected information, the problem becomes bigger when large, complex systems are considered. The methodology demonstrated in this work uses metamodel (such as a class diagram) and transformation rules (OCL - Object Constraint Language) to retrieve the sequence diagram of an implementation scenario.

6.2 Visualization and Documentation of Software

This section presents related work on software visualization and documentation which has become an important technique in understanding programs.

6.2.1 Visualization

Currently, there are many tools to visualize the structure, behavior and evolution of the program. In the paper of Maruyama, Omori and Hayashi (2014), a tool was developed for source code visualization, CodeForest. In this tool, the source code is seen as a forest in which each class is a tree. The tool CodeForest helps the user to experiment with a large number of combinations through mapping software on the visual metrics parameters. In addition, it records user actions taken in order to understand historical data that would accelerate the future tasks of software understanding.

Kobayashi et al. (2013) presented a paper showing how to facilitate comprehension of software architecture through the Sarf Map tool, a technique that visualizes software architecture in terms of its features. The Sarf Map tool visualizes implied features of software using a clustering algorithm based on software dependencies. A general map can be used to make level high decisions for reuse and also for communication between developers and other interested parts. In Sarf Map, each feature is displayed as blocks of a city, and classes are defined as buildings reflecting software layers. The relevance of features is represented as streets. Through studies on open source software, the architecture of the target systems could be easily displayed and the quality of the projects can be quickly assessed. However, the tool (KOBAYASHI et al., 2013) did not support all cases. For example, there may be features which are superimposed and present different points of view. In this sense, the tool does not support such situation. Other issue is the quality of dependency graph and scalability for large systems. A system with many classes makes viewing and management difficult. Finally, scalability also relates to colors, since humans can exhibit a shortcoming in discriminate colors, so when software has many packages, patterns of source organization are difficult to identify.

There are also techniques which retrieves software architecture organized in layers (BELLE et al., 2015). There are many different applied techniques, such as, clustering (SCHMIDT; MACDONELL; CONNOR, 2014), class grouping with common relations (SCANNIELLO et al., 2010), composition operations based on cohesion and coupling metrics and based static and dynamic information (ANDREOPOULOS et al., 2007).

The technique proposed in this thesis (Keecle) aim at reducing data visualization noise, through a data reduction process that occurs at each phase. In this thesis, we provide a visualization of key classes in a dependency graph. Thus, from a small set of classes it is easy to understand the dependency relationships and the hierarchical organization of classes.

6.2.2 Documentation

From the key classes we expect to approximate an architectural description that allows initial understanding of the main concepts of the system. Some studies (ROBILLARD; MEDVIDOVIĊ, 2016) show the deficiencies of documentation in open source systems and recommends a set of guidelines for the formulation of system documentation. This paper reported on an interview-based study of 18 authors of different chapters from the two-volume book "Architecture of Open-Source Applications". The main contributions are a synthesis of the process of authoring essay-style documents (ESDs) on software architecture, a series of observations on important factors that influence the content and presentation of architectural knowledge in this documentation form, as well as a set of recommendations for readers and writers of ESDs on software architecture. They analyzed the influence of three factors in particular: the evolution of a system, the community involvement in the project, and the personal characteristics of the author. The results showed a concern with accessibility of documentation.

The software architecture documentation concepts found in the above books are also discussed in many smaller publications that focus on architecture description languages ((CHAPMAN et al., 2010), (LEVESON, 1995), (MEDVIDOVIĊ; TAYLOR, 2000)), UML ((ALMORSY; GRUNDY; IBRAHIM, 2013) (LANGE; CHAUDRON; MUSKENS, 2006), (MEDVIDOVIC et al., 2002)), architectural patterns and styles ((BARNES; GARLAN; SCHMERL, 2014), (JANSEN; AVGERIOU; VEN, 2009), (OMMERING et al., 2000)), architectural views ((BRINKKEMPER; PACHIDI, 2010), (HEESCH; AVGERIOU; HILLIARD, 2012), (KRUCHTEN, 1995)), and standardized templates for capturing architectural knowledge ((GRAAF et al., 2012), (TANG; LIANG; VLIET, 2011)). The use of UML in practice was not widely used due to its complexity, lack of formal semantics, lack of inter-view synchronization, and the resulting inconsistencies.

A recent paper presented problems related to documentation manual nature of its creation and the gap between the creators and consumers (ROBILLARD et al., 2017). They discussed the major challenges they face in realizing such a paradigm shift, highlight existing research that can be leveraged to this end, and promote opportunities for increased convergence in research on software documentation.

A survey investigated the effectiveness of documentation for particular tasks (LETHBRIDGE; SINGER; FORWARD, 2003). In this study, only 35% of the developers reported use the documentation for software maintenance tasks whereas 61% reported the use of documentation for learning tasks related to software. So, they recognize the need to find ways to express the most useful information in less space and to make documentation easier to update, perhaps semi automatically.

A particular paper investigated the kind of information that a documentation should contain to support maintenance task (SOUZA; ANQUETIL; OLIVEIRA, 2005):

- differentiate four stages of experience (from newcomer, the first day of work; to expert, after some years of work on a system). For each stage, they propose different documents: newcomers need a short general view of the system; apprentices need the system architecture; interns need task oriented documents such as requirement description, process description, examples, step by step instructions; finally, experts need low level documentation as well as requirement description, and design specification;

- a document must describe the hierarchical architecture of the system;

- should include notes on the application domain, dependencies among classes, detailed description of a class' methods.

The available software documentation often contains information that does not match the reality of the code. A survey was presented in (LATOZA; MYERS, 2010) surveyed professional software developers and asked them to list hard-to-answer questions that they had recently asked about code. There are 179 respondents reporting on 371 questions. The most frequently reported question categories dealt with intent and rationale – *what does this code do, what is it intended to do, and why was it done this way?* Often, information needs of developers are difficult to address, such as questions about intent and rationale. Understanding information needs may provide clues to techniques for enabling developers to express needs; expression may be particularly difficult if developers have an incomplete or even incorrect understanding of the information they need.

In this thesis, we did not aimed to propose a new model for software documentation. A reason for generating information around the key classes is based on a study presented, showing that the main barrier for newcomers comprehending a software is the lack of documentation or if it does not contain organized and adequate content (PINTO; STEINMACHER; GEROSA, 2016) (STEINMACHER et al., 2013). Documentation information should be a guide for developers to comprehend and to perform distinct tasks during the software development, so design details is important to consider in a documentation. In addition, developers present distinct experience levels and interest on the documentation, so it is necessary establish a structure to attend different developer profiles.

In the software visualization context the most of studies address the problem of how much information is viewed and how it is organized. In this way, the amount of information presented constitutes a barrier for developers because results in large documentation that it difficult to find relevant information and requires developers to consult the source code directly. Keecle recovers dependency relationships and shows architecture organization in a dependency graph. In addition the organized information around key classes focuses on important design details to guide the developers understanding process.

6.3 Software Architecture Recovery

This section presents related work based on several techniques to extract software architecture (DUCASSE; POLLET, 2009a). In a recent study, Garcia, Ivkovic and Medvidovic̀ (2013b) performed an evaluation with six software architecture recovery techniques based on clustering, and here we review the two most relevant ones: ACDC and WAC.

ACDC technique (TZERPOS; HOLT, 2000) recovers components discovering clusters that follow patterns commonly observed in decomposition of large systems. It automatically assigns also meaningful names to clusters. The approach was evaluated with two case studies using stability and skeleton size. In the comparative analysis of Garcia, Ivkovic and Medvidovic̀, ACDC accuracy was confirmed, but in some cases, the tool can produce very small clusters. Keecle criteria benefit larger subtrees, and the limitation is encountered when small relevant subtrees are discarded. Fortunately, these situations seemed to be exceptions.

WCA technique (MAQBOOL; BABRI, 2004) is a clustering algorithm based on inter-cluster distance during the clustering process. They used code routines as clustering entities since they are able to reflect existing cohesion between entities. Evaluation of the approach with two systems is given in terms of recall and precision. Experiments with WCA (GARCIA; IVKOVIC; MEDVIDOVIC̀, 2013b) showed that for some inputs, the MoJoFM metric was not calculated because of memory errors.

In the paper from Garcia et al. (2011), a technique was designed to recover concerns associated with components to facilitate the understanding of the cluster meaning. Another kind of element recovered by the approach is connectors that describe the interaction between the components. The evaluation with eight systems produced reasonable results. A limitation of the approach it is the need of the users defining the number of clusters that provide the best result. In Keecle the number of key classes is defined from users interest to comprehend more or less details of the system design. Keecle is defined to recover a small set of classes that are central to comprehend the software architecture. Most of the software architecture recovery techniques based on clustering process are limited to retrieve classes and provide some type of organization such as components. Most of the techniques are aimed at decreasing the effort to understand such code elements, but the amount of classes and or components recovered by the approaches.

Another dynamic approach to recover architecture is Discotect (YAN et al., 2004) which filter important classes in the production of more abstract models. A case study with AAMS, a closed software, was conducted. One drawback is that the developer needs to specify an automaton that filters the desired classes.

A hybrid model that combines static analysis and dynamics for recovering architecture was shown in (RIVA; RODRIGUEZ, 2002). The main features are: visualization of static and dynamic views, synchronization of abstractions performed on the views, scripting support and

management of the use cases. The approach and the environment were demonstrated with only one example.

The study by Zaidman and Demeyer is the most closely related to ours (ZAIDMAN; DEMEYER, 2008). They proposed a technique to automatically identify so-called key classes of a software system that can be useful for a software engineer who is trying to get a high-level overview of a unfamiliar system. In this paper, the authors defined key classes as being typically characterized with having a lot of ”control“ within the application. Their technique is based on the identification of tightly coupled classes. They also take into account indirect coupling through the application of a webmining algorithm. In our work, software components are formed by large trace subtrees containing key classes.

A recent paper used centrality measures to identify the key classes in software systems (DING; LI; HE, 2016). The authors defined key classes as being key nodes that refers to a number of nodes which are more likely to affect the structure and function in a software network. In this paper, the authors presented a contribution based on four new measures based on the h-index to study class importance, according to the degree of neighbors and the edge weights and compared with several existing centrality measures. The authors validated the feasibility of proposed measures to identify important nodes. The approach was validated on three open source software (Tomcat, Ant and JUNG) using version control log derived by TortoiseSVN for each software system. The result indicates that the classes with greater centrality are changed more frequently. The proposed measures not only are able to identify the key classes as some commonly used centrality measures (correlative coefficient 0.987) but also perform better than some commonly used centrality measures (the improvement is at least 0.215). However, their approach is not adequate for small set of key classes, so when for instance $k = 5$ the most important five classes have not been modified, but when $k = 200$, these classes were successfully recognized that they needed to be changed. We believe that the number of key classes depends of the interest of the developer in understanding the design with more or less details and mainly that key classes must be a guide for design comprehension, but, key classes large set is a barrier that difficult the design comprehension by newcomers.

Another recent paper related to Zaidman and Demeyer (2008) is discussed. The authors proposed to model softwares as a network, where the classes are the vertices in the network and the dependencies are the edges, and apply K-core decomposition to identify a core subset of vertices as potentially important classes (key classes) (MEYER; SIY; BHOWMICK, 2014). They studied three open source Java projects over a 10-year period and demonstrate, using different metrics, that the K-core decomposition of the network can help us identify the key classes of the corresponding software. The authors compared their approach with Zaidman and Demeyer (2008) for Apache Ant software. Zaidman and Demeyer identified the top 10 classes in Apache Ant while the approach presented by MEYER, SIY and BHOWMICK (2014) returned similar results, a factor that limit the approach is related to low precision: 64 key classes were

recovered. Our approach recovers the desired number of key classes by developers alleviating the barriers for design comprehension. We used dynamic dependencies to mine key classes, but dynamic analysis eliminates abstract classes and interfaces which serve as useful starting points for concept location in program comprehension. We still could rely on manual analysis of classes extending or implementing abstract classes and interfaces to potentially include them in result set.

Another paper was proposed a tool to automatically extract such a summary, by identifying the most important classes of the system (SORA, 2015). In this paper, the importance of a class (key class) is given by the amount and types of interactions it has with other classes. Thus, a natural approach of identifying the most important classes was based on ranking them with a graph-ranking algorithm. This approach consists of modeling the static dependencies of the system as a graph and applying a graph ranking algorithm. They empirically determined how different dependency types should be taken into account in building the system graph. They compared the results using Apache Ant in relation to work Zaidman and Demeyer (2008) around 30 classes were mined, presenting recall similar to Zaidman and Demeyer.

6.4 Structural Properties - Smells

This section presents related work on bad smells. Research on architectural smells to solve design problems have gained increased interest. Garcia et al. (2009a) investigated the concept of architectural ”bad smells”, which are recurring results of software decisions that can have non-obvious and significant detrimental effects on system lifecycle properties. The authors presented architectural smells and differentiate them from related concepts, such as architectural antipatterns and code smells. In that paper, the authors only describe four representative architectural smells, a experimental study was not reported. Subsequently, this approach was considered in an experiment evolving two large industrial systems (GARCIA et al., 2009b). The experience gained indicates the need to identify and catalog architectural smells so that software architects can discover and eliminate these from system design. The proposed approach in this thesis can be useful in the sense that more important architectural smells are likely to be found in key classes.

Oizumi et al. (2014) studied code-anomaly agglomerations. An agglomeration is a group of code anomalies that are related to each other for some reason and affects syntactically-related code elements in the program. The results showed that architectural problems are much more often reflected as anomaly agglomerations rather than individual anomalies in the source code. The approach considered a total of 5418 code anomalies and 2229 agglomerations within 7 systems. They conclude that agglomerations have a higher chance to be related to architectural problems than individual anomalies as they naturally affect more code elements than individual code anomalies. Nonetheless, the results in this thesis suggest that individual anomalies on

key classes also have higher chances of being related to architectural problems because of the inherent nature of architecturally relevant classes.

Some authors have studied the lifespan of code smells using software repositories (PETERS; ZAIDMAN, 2012). The idea is to provide insight into the behavior of software developers with regard to resolving code smells and anti-patterns. In one particular case study, these researchers investigated the lifespan of code smells and the refactoring behavior of developers in seven open source systems. The results of this study indicated that engineers are aware of code smells, but are not very concerned with their impact, given the low refactoring activity. A problem that affects the results is related to the deliberate refactorings that must be distinguished from other coding activities that coincidentally result in the removal of a code smell. Log messages have to be inspected manually to make this distinction, which is usually clear, taking the aforementioned threat into account. Indeed, with our results, smells found in key classes seems to have higher priority over smells in non-key classes, and refactoring activities on key classes may have important impact in the overall architecture.

Similar papers were presented (LOZANO; WERMELINGER; NUSEIBEH, 2007) (PALOMBA et al., 2015), (PALOMBA et al., 2013). The existence of bad smells points to important design problems which in turn help to focus developers' efforts. The authors performed an empirical study of the evolution of software systems and limited their studies to particular kinds of smells.

A study to investigate if classes with code smells are more change-prone than classes without smells was presented in (KHOMH; PENTA; GUEHENEUC, 2009). The authors performed an case study with releases of Eclipse and Azureus and showed that classes with code smells are more change-prone than the others. Other recent studies (TUFANO et al., 2015) conducted a large empirical study over the change history of 200 open source projects from different software ecosystems and investigated when bad smells are introduced by developers, and the circumstances and reasons behind their introduction. Their study required a strategy to identify smell introducing commits. The paper revealed that most of the time code artifacts are affected by bad smells since their creation and newcomers are not necessarily responsible for introducing bad smells, while developers with high workloads and release pressure are more prone to introducing smell instances. Our results showed that main owners of key classes are also responsible for a majority of commits and so, they would be equally likely to be responsible for smells around in key classes.

These approaches use a limited number of bad smells and are sensitive to accuracy of smell detection tools. Our approach also depends of smell detection tools, but the accuracy impact is alleviated because we are working with a reduced set of key classes. Our goal was to increase the architectural learning using some relevant classes and to analyze if they are more prone to bad smells and if the owner of the key class is more prone to introducing a bad smell in the code.

Keecle recovers dependency relationships and shows architecture organization in a depen-

dency graph. A study showed that problematic architectural connections can propagate errors (XIAO, 2015). The architectural connections that violate common design principles strongly correlate with the error-proneness of files. This can become debts that accumulate high interest in terms of maintenance costs over time. A quantitative model for project managers and stakeholders as a reference in making decisions of whether, when and where to invest in refactoring is presented. Another work that investigates the importance of analyzing the software dependencies are treated (NORD et al., 2014). The motivation is to understand rework costs for safety-critical systems.

6.5 Social Properties - Ownership

The empirical studies based on related work have been conducted in order to understand and leverage different aspects of commits will be presented in more detail. The analyzed papers reinforce the evidence found in this thesis related to design problems caused by ownership.

Similar to the results achieved in the proposal of Geldenhuys (2010) evaluated the rule which states that 20% of developers contribute 80% of the work on a project (MOCKUS; FIELDING; HERBSLEB, 2000), (MOCKUS; FIELDING; HERBSLEB, 2002) and (VIR; MING; TAN, 2007). To evaluate this behavior they analyzed the number of contributions, duration of involvement with the project, and the number of modifications to source code files. The results showed that it is not a trivial procedure to identify who are the core developers of a project and therefore, this rule is not very well defined, contrary to what the literature suggests. The evaluated projects do not have the same developer engagement standards.

Kolassa, Riehle and Salim (2013) evaluated the frequency of commits of an developer, as a fundamental aspect to understand the process of software development. According to the authors the results are useful for evaluating the workload of developers and if the developers are productive by checking whether they contribute regularly. Eyolfson, Tan and Lam (2011) analyzed 57,028 and 4,399 bug-fixing commits in two large and widely-used open-source software projects, the Linux kernel and PostgreSQL, to study the correlation between commit correctness with several commit social characteristics, such as the time-of-day of commits, the day-of-week of commits, developer experience, and developers' commit frequency. The authors presented several interesting findings, including: (1) late-night commits (between midnight and 4:00 AM) are buggier than average, while morning commits (7:00 AM–noon) are less buggy.

The work of ownership can provide information regarding the quality of the code. In (BIRD et al., 2011) show that ownership are responsible by inserting fault, this fact is also shown in the work (TUFANO et al., 2015), indicating that contrary to what many think bugs are introduced in the initial stages of development as a result of the high workloads of ownership.

In this thesis we show that ownership of key classes is low when compared to other classes of software. However, this evidence does not alleviate the amount of structural problems en-

countered in key classes.

6.6 Concluding Remarks

In this chapter, we selected studies to better understand the state of the art related to this thesis.

Firstly, we presented studies on program comprehension (Section 6.1) centered on execution traces highlighting their problems, such as, related to size and their visualization in sequence diagrams.

In Section 6.2, we showed the studies related to software visualization and documentation. The studies highlighted for instance, software architecture visualization in terms of its features and concerning to documentation, the studies highlighted the deficiencies of the documentation in open source systems.

In Section 6.3, we presented some techniques to software architecture recovery such as ACDC, WCA, Discotect, and their limitation when compared with our approach. In addition we highlighted four studies considering recover key classes.

In Section 6.4, we showed studies relevant concerning to such as architectural bad smells and if classes with code smells are more change-prone than classes without smells.

Finally, in Section 6.5 we highlighted some studies related to ownership information.

Next chapter, we will present the conclusion and future work related to this thesis.

Chapter 7

Conclusion

Key classes have been an alternative adopted by many OSS to generate documentation. However, there is no widely adopted criteria for obtaining them, resulting in a set of classes that may have some variation in interest among the developers. Thus, the documentation based on key classes is tailored to support a specific need of developers: understanding and assessing design.

We have developed a semi-automatic approach to mine a set of key classes, whose size is indicated by the developer. The results in terms of recall and precision were promising compared to previous work. A set of steps were taken to reduce the amount of information available from the source code: a) capture execution traces from execution scenarios; b) compression of traces; c) trace subtree extraction; d) elimination of identical subtrees traces; e) classification of trace subtree; and f) extraction of roots from subtrees to obtain key classes. We adopted dynamic analysis as a natural way of reducing the high volume of information generated by traces. The important software features have to be exercised, to achieve a reasonable coverage of the classes of the software, that is important to reduce an adequate set of key classes. Although dynamic analysis only captures concrete classes of the target software, in our final result through a manual analysis we also consider manual introduction of abstract classes and interfaces that were implemented or extended by concrete classes.

However, a simple set of key classes is not able to move developers over the needs of considering them during maintenance activities. The set of key classes tends to be small and so, is adequate as a starting point to understand software design and a guide to support maintenance tasks.

To make developers aware of the importance of key classes to evaluate software design, in a static context, we performed a study of the social and structural properties that involve them. In this study, we observed that the key classes present more problems in relation to the non-key classes of the system. Three properties were investigated: dependency graphs, the ownership involvement degree and the relation of the cohesion and coupling metrics with bad smells. The results showed that:

- A meaningful structured view of key classes with respective violations can be produced, suggesting that developers would benefit from that in situations where the software documentation is not available, or supplementing current documentation with additional information about dependencies.
- Key classes have more specific smells when compared to non-key classes. Also, for coupling and cohesion metrics for key classes are significantly worse than for non-key classes. The use key class information can lead to a more focused way to find relevant design anomalies supported by the design nature of key classes.
- Ownership in key classes has a lower level compared to non-key classes. The number of main owners seems to be reduced, either for key and non-key classes, suggesting that prioritizing code review code of owners when committing in key classes would produce more benefits in design.

We performed two studies involving developers and students. Overall, among the evaluated criteria, there were no significant differences between the groups (Key classes, Traditional or Key classes + Traditional), suggesting that:

- Considering the usefulness of the documentation based on key classes, the developers presented a positive perspective, which could be a result of the reasonable obtained recall and precision obtained. Another positive aspect from the developers and students was to suggest that key classes and other resources such as trace trees, dependency graph, etc., are an important starting point to comprehend the application design.
- Key classes can to produce relevant information to guide developers in maintenance activities, introduction of new features, bug fixes, complexity metrics and bad smells information and can solve application structural problems.
- Another important information observed was the positive opinion of the developers and students about the possibility of the documentation based on key classes to complement original documentation satisfaction criterion because it organizes the system structure using diagram instead on only textual descriptions long.

Therefore, our objectives were fulfilled when we presented a new semi-automated approach capable of generating design information from a reduced set of key classes, and when organized into documentation based on key classes produce results equivalent to those of traditional documentation.

Finally, we envision two possible future works related to the approach proposed in this thesis.

Key classes alignment: In this future work, one could investigate if aligning key classes several applications in the same domain, we could get concepts that could help us get reference

classes. For example, considering the key classes of Tomcat[1], Jetty[2] and Undertow[3], when aligned semantically to evaluate if a class in one system does a similar thing of other class in another system, i.e, if they the same concept. Then, if we can align the key classes, then they implement important concepts with respect to a reference class. Moreover, we could analyze whether a key class implements some extra feature to include in another system. So, the idea is to evaluate if those classes match with the other classes to see how the incorporation of those classes would be possible as a reference class and which is the coupling requirement that should be implemented to take those classes to other systems. This would be a productive way to enhance an application.

Large scale refactoring: Another area of future work is to investigate how large scale refactoring affects key classes. This can be used to identify when large refactorings have occurred in the history of the system and how they were implemented, which in turn can help guide the way future refactorings are conducted.

Design quality evaluation: Another area of future work is to investigate if a bad design can mask key classes, and prevent the proposed approach generates relevant results. In this case, approach can evolve to detect the quality level of a design according to the difficulty in obtaining the key classes. Finally, this evaluation and be useful to insert points of a design that must suffer refactoring.

Keecle for another programming language: Another area of future work is to extend Keecle to extract key classes in other systems implemented in distinct programming languages such as language C.

Occurrence of bugs in key classes: Key classes provide a more focused way to find more relevant design anomalies classes supported by the design nature of key classes. But, it is necessary more studies to investigate. So, as a future work, for example a study that in fact take more the occurrence of bugs on key classes in in relation to non-keys classes.

Ease of understanding: Perform further studies with human subjects showing that the documentation based on key classes facilitates understanding object oriented system design compared to traditional documentation. In this case, we want to know if key classes can be used to discover certain aspects in the architecture, such as architectural patterns. Another possibility, is to evaluate the degree of correctness of the task (which would give to understand why the documentation helped or not). For example, involving questions about design (what architectural pattern is used, what are the main services performed and their collaborations, etc.)

Template for documentation based on key classes: There is no design documentation template widely accepted by the developers community. This may be a barrier to understanding software design. As future work, it is proposed the investigation of new resources centered in

[1] http://tomcat.apache.org/
[2] http://www.eclipse.org/jetty/
[3] http://undertow.io/

the design for documentation based on key classes that accelerate the learning process of the software.

Bibliography

ALMORSY, M.; GRUNDY, J.; IBRAHIM, A. S. Automated software architecture security risk analysis using formalized signatures. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 662–671. ISBN 978-1-4673-3076-3. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486875>.

ANDREOPOULOS, B. et al. Clustering large software systems at multiple layers. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 49, n. 3, p. 244–254, Mar. 2007. ISSN 0950-5849.

ASTUDILLO, H.; VALDES, G.; BECERRA, C. Empirical measurement of automated recovery of design decisions and structure. In: **Proceedings of the 2012 Andean Region International Conference**. Washington, DC, USA: IEEE Computer Society, 2012. (ANDESCON '12), p. 105–108. ISBN 978-0-7695-4882-1.

BALL, T.; EICK, S. G. Software visualization in the large. **Computer**, v. 29, n. 4, p. 33–43, 1996. ISSN 0018-9162.

BARNES, J. M.; GARLAN, D.; SCHMERL, B. Evolution styles: Foundations and models for software architecture evolution. **Softw. Syst. Model.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 13, n. 2, p. 649–678, May 2014. ISSN 1619-1366.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0-201-19930-0.

BELLE, A. B. et al. The layered architecture recovery as a quadratic assignment problem. In: **ECSA**. [S.l.]: Springer, 2015. (Lecture Notes in Computer Science, v. 9278), p. 339–354. ISBN 978-3-319-23726-8.

BIRD, C. et al. Don't touch my code!: Examining the effects of ownership on software quality. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 4–14. ISBN 978-1-4503-0443-6.

BRIAND, L. C.; LABICHE, Y.; LEDUC, J. Toward the reverse engineering of uml sequence diagrams for distributed java software. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 32, n. 9, p. 642–663, Sep. 2006. ISSN 0098-5589.

BRINKKEMPER, S.; PACHIDI, S. Functional architecture modeling for the software product industry. In: **Proceedings of the 4th European Conference on Software Architecture**. Berlin, Heidelberg: Springer-Verlag, 2010. (ECSA'10), p. 198–213. ISBN 3-642-15113-2, 978-3-642-15113-2.

BROOKS, F. P. **The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition**. [S.l.]: Addison-Wesley Professional, 1995.

BROOKS, R. Towards a theory of the comprehension of computer programs. **International Journal of Man-Machine Studies**, v. 18, n. 6, p. 543 – 554, 1983. ISSN 0020-7373.

BROWN, W. J. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. New York, NY, USA: John Wiley & Sons, Inc., 1998. ISBN 0-471-19713-0.

CHAPARRO, O. et al. On the impact of refactoring operations on code quality metrics. In: **30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014**. [s.n.], 2014. p. 456–460. Available: <http://dx.doi.org/10.1109/ICSME.2014.73>.

CHAPMAN, C. et al. Software architecture definition for on-demand cloud provisioning. In: **Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing**. New York, NY, USA: ACM, 2010. (HPDC '10), p. 61–72. ISBN 978-1-60558-942-8.

CHARALAMPIDOU, S.; AMPATZOGLOU, A.; AVGERIOU, P. Size and cohesion metrics as indicators of the long method bad smell: An empirical study. In: **Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: ACM, 2015. (PROMISE '15), p. 8:1–8:10. ISBN 978-1-4503-3715-1.

CHESS, B.; MCGRAW, G. Static Analysis for Security. **IEEE Security and Privacy**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 2, n. 6, p. 76–79, Nov. 2004. ISSN 1540-7993.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994. ISSN 0098-5589.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: a taxonomy. **IEEE Software**, v. 7, n. 1, p. 13–17, Jan 1990. ISSN 0740-7459.

CORNELISSEN, B. et al. A systematic survey of program comprehension through dynamic analysis. **Software Engineering, IEEE Transactions on**, v. 35, n. 5, p. 684–702, Sept 2009. ISSN 0098-5589.

CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: **2011 International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2011. p. 275–284. ISSN 1949-3770.

DEBBARMA, M. K. et al. A review and analysis of software complexity metrics in structural testing. p. Vol–2, 2013.

DELINE, R. et al. Towards understanding programs through wear-based filtering. ACM, New York, NY, USA, p. 183–192, 2005.

DING, Y.; LI, B.; HE, P. An improved approach to identifying key classes in weighted software network. v. 2016, p. 1–9, 01 2016.

DUCASSE, S.; POLLET, D. Software architecture reconstruction: A process-oriented taxonomy. **IEEE Trans. Software Eng.**, v. 35, n. 4, p. 573–591, 2009.

____. Software architecture reconstruction: A process-oriented taxonomy. **IEEE Trans. Software Eng.**, v. 35, n. 4, p. 573–591, 2009.

DUGERDIL, P. Using trace sampling techniques to identify dynamic clusters of classes. In: **Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research**. Riverton, NJ, USA: IBM Corp., 2007. (CASCON '07), p. 306–314.

EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Aiding program comprehension by static and dynamic feature analysis. p. 602–611, 2001. ISSN 1063-6773.

____. Locating features in source code. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 29, n. 3, p. 210–224, Mar. 2003. ISSN 0098-5589.

EYOLFSON, J.; TAN, L.; LAM, P. Do time of day and developer experience affect commit bugginess? In: **Proceedings of the 8th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2011. (MSR '11), p. 153–162. ISBN 978-1-4503-0574-7.

FONTANA, F. A. et al. An experience report on using code smells detection tools. p. 450–457, 2011.

FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-48567-2.

FROEHLICH, J.; DOURISH, P. Unifying artifacts and activities in a visual tool for distributed software development teams. p. 387–396, 2004. ISSN 0270-5257.

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIĊ, N. A comparative analysis of software architecture recovery techniques. In: **ASE**. [S.l.]: IEEE, 2013. p. 486–496.

____. A comparative analysis of software architecture recovery techniques. In: **ASE**. [S.l.]: IEEE, 2013. p. 486–496.

GARCIA, J. et al. Obtaining ground-truth software architectures. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 901–910. ISBN 978-1-4673-3076-3.

____. Identifying architectural bad smells. In: **Proceedings of the 2009 European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2009. (CSMR '09), p. 255–258. ISBN 978-0-7695-3589-0.

____. Toward a catalogue of architectural bad smells. In: **Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems**. Berlin, Heidelberg: Springer-Verlag, 2009. (QoSA '09), p. 146–162. ISBN 978-3-642-02350-7.

____. Enhancing architectural recovery using concerns. In: **Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 552–555. ISBN 978-1-4577-1638-6.

GELDENHUYS, J. Finding the core developers. **2013 39th Euromicro Conference on Software Engineering and Advanced Applications**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 447–450, 2010.

GRAAF, K. A. de et al. Ontology-based software architecture documentation. In: **Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture**. Washington, DC, USA: IEEE Computer Society, 2012. (WICSA-ECSA '12), p. 121–130. ISBN 978-0-7695-4827-2.

GREEVY, O.; DUCASSE, S. Correlating features and code using a compact two-sided trace analysis approach. In: **CSMR**. [S.l.]: IEEE Computer Society, 2005. p. 314–323. ISBN 0-7695-2304-8.

HAMOU-LHADJ, A.; LETHBRIDGE, T.; FU, L. Challenges and requirements for an effective trace exploration tool. p. 70–78, 2004.

HAN, J. **Data Mining: Concepts and Techniques**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1558609016.

HEESCH, U. van; AVGERIOU, P.; HILLIARD, R. A documentation framework for architecture decisions. **Journal of Systems and Software**, v. 85, n. 4, p. 795 – 820, 2012. ISSN 0164-1212. Available: <http://www.sciencedirect.com/science/article/pii/S0164121211002755>.

HEUZEROTH, D. et al. Automatic design pattern detection. In: **Proceedings of the 11th IEEE International Workshop on Program Comprehension**. Washington, DC, USA: IEEE Computer Society, 2003. (IWPC '03), p. 94–. ISBN 0-7695-1883-4.

____. Automatic design pattern detection. IEEE Computer Society, p. 94–104, 2003.

HEUZEROTH, D.; HOLL, T.; LOWE, W. Combining static and dynamic analyses to detect interaction patterns. p. 2, 2002.

HUPFER, S. et al. Introducing collaboration into an application development environment. ACM, New York, NY, USA, p. 21–24, 2004.

JANSEN, A.; AVGERIOU, P.; VEN, J. S. van der. Enriching software architecture documentation. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 82, n. 8, p. 1232–1248, Aug. 2009. ISSN 0164-1212.

JERDING, D.; RUGABER, S. Using visualization for architectural localization and extraction. p. 56–65, Oct 1997.

KAKARONTZAS, G. et al. Layer assessment of object-oriented software. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 2, p. 349–366, Feb. 2013. ISSN 0164-1212.

KAZMAN, R.; CARRIÈRE, S. J. Playing detective: Reconstructing software architecture from available evidence. **Automated Software Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 6, n. 2, p. 107–138, Apr. 1999. ISSN 0928-8910.

KERSTEN, M.; MURPHY, G. C. Mylar: A degree-of-interest model for ides. ACM, New York, NY, USA, p. 159–168, 2005. Available: <http://doi.acm.org/10.1145/1052898.1052912>.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: **Proceedings of the 2009 16th Working Conference on Reverse Engineering**. [S.l.: s.n.], 2009. (WCRE '09), p. 75–84. ISBN 978-0-7695-3867-9.

KOBAYASHI, K. et al. Sarf map: Visualizing software architecture from feature and layer viewpoints. In: **ICPC**. [S.l.: s.n.], 2013. p. 43–52.

KOLASSA, C.; RIEHLE, D.; SALIM, M. A. The empirical commit frequency distribution of open source projects. In: **Proceedings of the 9th International Symposium on Open Collaboration**. New York, NY, USA: ACM, 2013. (WikiSym '13), p. 18:1–18:8. ISBN 978-1-4503-1852-5.

KOSKINEN, J.; KETTUNEN, M.; SYSTA, T. Profile-based approach to support comprehension of software behavior. p. 212–224, 2006. ISSN 1092-8138.

KRUCHTEN, P. The 4+1 view model of architecture. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 6, p. 42–50, Nov. 1995. ISSN 0740-7459.

KRUSKAL, W. H.; WALLIS, W. A. Use of Ranks in One-Criterion Variance Analysis. **Journal of the American Statistical Association**, American Statistical Association, n. 260, p. 583–621. ISSN 01621459.

LANGE, C. F. J.; CHAUDRON, M. R. V.; MUSKENS, J. In practice: Uml software architecture and design description. **IEEE Software**, v. 23, n. 2, p. 40–46, March 2006. ISSN 0740-7459.

LANZA, M.; DUCASSE, S. A categorization of classes based on the visualization of their internal structure: The class blueprint. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 36, n. 11, p. 300–311, Oct. 2001. ISSN 0362-1340.

LATOZA, T. D.; MYERS, B. A. Hard-to-answer questions about code. In: **Evaluation and Usability of Programming Languages and Tools**. New York, NY, USA: ACM, 2010. (PLATEAU '10), p. 8:1–8:6. ISBN 978-1-4503-0547-1. Available: <http://doi.acm.org/10.1145/1937117.1937125>.

LETHBRIDGE, T. C.; SINGER, J.; FORWARD, A. How software engineers use documentation: The state of the practice. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 6, p. 35–39, Nov. 2003. ISSN 0740-7459.

IEEE Trans. Softw. Eng., IEEE Press, Piscataway, NJ, USA, v. 21, n. 4, 1995. ISSN 0098-5589.

LOZANO, A.; WERMELINGER, M.; NUSEIBEH, B. Assessing the impact of bad smells using historical information. In: **Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting**. [S.l.: s.n.], 2007. (IWPSE '07), p. 31–34. ISBN 978-1-59593-722-3.

MAALEJ, W. et al. On the comprehension of program comprehension. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 31:1–31:37, Sep. 2014. ISSN 1049-331X.

MäNTYLä, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, n. 3, p. 395–431, Sep. 2006. ISSN 1382-3256. Available: <http://dx.doi.org/10.1007/s10664-006-9002-8>.

MAQBOOL, O.; BABRI, H. A. The weighted combined algorithm: A linkage algorithm for software clustering. In: **CSMR**. [S.l.]: IEEE Computer Society, 2004. p. 15–24. ISBN 0-7695-2107-X.

MARCUS, A.; FENG, L.; MALETIC, J. I. Comprehension of software analysis data using 3d visualization. p. 105–114, 2003. ISSN 1092-8138.

MARUYAMA, K.; OMORI, T.; HAYASHI, S. A visualization tool recording historical data of program comprehension tasks. In: **Proceedings of the 22Nd International Conference on Program Comprehension**. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 207–211. ISBN 978-1-4503-2879-1.

MEDVIDOVIC, N. et al. Modeling software architectures in the unified modeling language. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 11, n. 1, p. 2–57, Jan. 2002. ISSN 1049-331X.

MEDVIDOVIĊ, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. **IEEE Transactions on Software Engineering**, v. 26, n. 1, p. 70–93, 2000. ISSN 0098-5589.

MEYER, P.; SIY, H.; BHOWMICK, S. Identifying important classes of large software systems through k-core decomposition. **Advances in Complex Systems**, v. 17, n. 07n08, p. 1550004, 2014. Available: <http://www.worldscientific.com/doi/abs/10.1142/S0219525915500046>.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. A case study of open source software development: The apache server. In: **Proceedings of the 22Nd International Conference on Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE '00), p. 263–272. ISBN 1-58113-206-9.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two case studies of open source software development: Apache and mozilla. **ACM Trans. Softw. Eng. Methodol.**, ACM, v. 11, n. 3, p. 309–346, Jul. 2002. ISSN 1049-331X.

MOHA, N. et al. Decor: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, v. 36, n. 1, 2010.

MÜLLER, H. A. et al. Reverse engineering: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE '00), p. 47–60. ISBN 1-58113-253-0.

MÜLLER, H. A.; KLASHINSKY, K. Rigi-a system for programming-in-the-large. IEEE Computer Society Press, Los Alamitos, CA, USA, p. 80–86, 1988.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: Bridging the gap between source and high-level models. ACM, New York, NY, USA, p. 18–28, 1995.

NAGAPPAN, N.; MURPHY, B.; BASILI, V. The influence of organizational structure on software quality: An empirical case study. In: **Proceedings of the 30th International Conference on Software Engineering**. [S.l.: s.n.], 2008. (ICSE '08), p. 521–530. ISBN 978-1-60558-079-1.

NORD, R. L. et al. Architectural dependency analysis to understand rework costs for safety-critical systems. In: **Companion Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE Companion 2014), p. 185–194. ISBN 978-1-4503-2768-8.

OIZUMI, W. et al. When code-anomaly agglomerations represent architectural problems? an exploratory study. In: **Software Engineering (SBES), 2014 Brazilian Symposium on**. [S.l.: s.n.], 2014. p. 91–100.

OJA, H. Robust nonparametric statistical methods, second edition by thomas p. hettmansperger, joseph w. mckean. v. 79, p. 300–300, 08 2011.

OMMERING, R. van et al. The koala component model for consumer electronics software. **Computer**, v. 33, n. 3, p. 78–85, Mar 2000. ISSN 0018-9162.

O'REILLY, C.; BUSTARD, D.; MORROW, P. The war room command console: Shared visualizations for inclusive team coordination. ACM, New York, NY, USA, p. 57–65, 2005.

PALOMBA, F. et al. Detecting bad smells in source code using change history information. In: **2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013**. [S.l.: s.n.], 2013. p. 268–278.

____. Mining version histories for detecting code smells. **IEEE Trans. Software Eng.**, v. 41, n. 5, p. 462–489, 2015.

PATEL, C.; HAMOU-LHADJ, A.; RILLING, J. **Software Clustering Using Dynamic Analysis and Static Dependencies**. 2009.

PAUW, W. et al. Software visualization: International seminar dagstuhl castle, germany, may 20–25, 2001 revised papers. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 151–162, 2002. Available: <http://dx.doi.org/10.1007/3-540-45875-1_12>.

PAUW, W. D. et al. Visualizing the execution of java programs. Springer-Verlag, London, UK, UK, p. 151–162, 2002. Available: <http://dl.acm.org/citation.cfm?id=647382.724791>.

PENNY, D. A. **The Software Landscape: A Visual Formalism for Programming-in-the-large**. Phd Thesis (PhD Thesis), Toronto, Ont., Canada, Canada, 1993. UMI Order No. GAXNN-82707.

PETERS, R.; ZAIDMAN, A. Evaluating the lifespan of code smells using software repository mining. In: **Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2012. (CSMR '12), p. 411–416. ISBN 978-0-7695-4666-7.

PINTO, G.; STEINMACHER, I.; GEROSA, M. A. More common than you think: An in-depth study of casual contributors. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 112–123.

PINZGER, M.; NAGAPPAN, N.; MURPHY, B. Can developer-module networks predict failures? In: **Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2008. (SIGSOFT '08/FSE-16), p. 2–12. ISBN 978-1-59593-995-1.

PISTOIA, M. et al. A survey of static analysis methods for identifying security vulnerabilities in software systems. **IBM Systems Journal**, v. 46, n. 2, p. 265–288, 2007. ISSN 0018-8670.

PRUIJT, L.; KÖPPE, C.; BRINKKEMPER, S. On the accuracy of architecture compliance checking support accuracy of dependency analysis and violation reporting. In: **IEEE 21st International Conference on Program Comprehension, ICPC**. [S.l.: s.n.], 2013. p. 172–181.

PRUIJT, L.; KP̈PE, C.; BRINKKEMPER, S. On the accuracy of architecture compliance checking support accuracy of dependency analysis and violation reporting. p. 172–181, 2013. ISSN 1063-6897.

REISS, S. P. An overview of BLOOM. ACM, p. 2–5, 2001.

RICHNER, T.; DUCASSE, S. Using dynamic information for the iterative recovery of collaborations and roles. p. 34–43, 2002. ISSN 1063-6773.

RIVA, C.; RODRIGUEZ, J. V. Combining static and dynamic views for architecture reconstruction. In: **Proceedings of the Sixth European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2002. (CSMR '02), p. 47–.

ROBILLARD, M. P. et al. On-demand developer documentation. **International Conference on Software Maintenance and Evolution**, IEEE, 2017.

ROBILLARD, M. P.; MEDVIDOVIC̀, N. Disseminating architectural knowledge on open-source projects. In: **Proceedings of the 38th ACM/IEEE International Conference on Software Engineering**. [S.l.: s.n.], 2016. p. 12.

ROBILLARD, M. P.; MURPHY, G. C. Feat a tool for locating, describing, and analyzing concerns in source code. p. 822–823, 2003. ISSN 0270-5257.

SARTIPI, K. Pattern-based software architecture recovery. In: **In Proc. of the Second ASERC Workshop on Software Architecture**. [S.l.: s.n.], 2003.

SARTIPI, K.; DEZHKAM, N. An amalgamated dynamic and static architecture reconstruction framework to control component interactions 259. In: **Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on**. [S.l.: s.n.], 2007. p. 259–268. ISSN 1095-1350.

SCANNIELLO, G. et al. Architectural layer recovery for software system understanding and evolution. **Softw., Pract. Exper.**, v. 40, n. 10, p. 897–916, 2010.

SCHMIDT, F.; MACDONELL, S. G.; CONNOR, A. M. An automatic architecture reconstruction and refactoring framework. **CoRR**, abs/1407.6103, 2014.

SHNEIDERMAN, B.; MAYER, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. **International Journal of Parallel Programming**, 1979.

SOBREIRA, V.; MAIA, M. d. A. A visual trace analysis tool for understanding feature scattering. In: **2008 15th Working Conference on Reverse Engineering**. [S.l.: s.n.], 2008. p. 337–338. ISSN 1095-1350.

SORA, I. Helping program comprehension of large software systems by identifying their most important classes. In: **Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers**. [s.n.], 2015. p. 122–140. Available: <https://doi.org/10.1007/978-3-319-30243-0_7>.

SOUZA, S. C. B. de; ANQUETIL, N.; OLIVEIRA, K. M. de. A study of the documentation essential to software maintenance. In: **Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information**. New York, NY, USA: ACM, 2005. (SIGDOC '05), p. 68–75. ISBN 1-59593-175-9. Available: <http://doi.acm.org/10.1145/1085313.1085331>.

STEINMACHER, I. et al. Overcoming open source project entry barriers with a portal for newcomers. In: **Proc. of the 38th Intl. Conf. on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 273–284. ISBN 978-1-4503-3900-1.

____. Why do newcomers abandon open source software projects? In: **2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)**. [S.l.: s.n.], 2013. p. 25–32.

STOREY, M.-A. Designing a software exploration tool using a cognitive framework of design elements. Springer, p. 113–148, 2003. Available: <http://www.amazon.com/exec/obidos/ASIN/1402074484>.

STOREY, M. A. Theories, methods and tools in program comprehension: past, present and future. p. 181–191, 2005. ISSN 1092-8138.

STOREY, M.-A. et al. Improving the usability of eclipse for novice programmers. ACM, New York, NY, USA, p. 35–39, 2003.

SYSTÄ, T.; KOSKIMIES, K.; MÜLLER, H. A. Shimba - an environment for reverse engineering java software systems. **Softw., Pract. Exper.**, v. 31, n. 4, p. 371–394, 2001. Available: <http://dblp.uni-trier.de/db/journals/spe/spe31.html#SystaKM01>.

TAHVILDAR, L.; KONTOGIANNIS, K. Improving design quality using meta-pattern transformations: A metric-based approach: Research articles. **J. Softw. Maint. Evol.**, John Wiley & Sons, Inc., New York, NY, USA, v. 16, n. 4-5, p. 331–361, Jul. 2004. ISSN 1532-060X.

TANG, A.; LIANG, P.; VLIET, H. v. Software architecture documentation: The road ahead. In: **Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on**. [S.l.: s.n.], 2011. p. 252–255.

TAYLOR, R. N.; MEDVIDOVIĊ, N.; DASHOFY, E. M. **Software Architecture: Foundations, Theory, and Practice**. [S.l.]: Wiley Publishing, 2009. ISBN 0470167742, 9780470167748.

TRUMPER, J.; DOLLNER, J.; TELEA, A. Multiscale visual comparison of execution traces. In: **Program Comprehension (ICPC), 2013 IEEE 21st International Conference on**. [S.l.: s.n.], 2013. p. 53–62. ISSN 1063-6897.

TUFANO, M. et al. When and why your code starts to smell bad. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 403–414. ISBN 978-1-4799-1934-5.

TZERPOS, V.; HOLT, R. C. Acdc : An algorithm for comprehension-driven clustering. In: **In Proceedings of the Seventh Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2000. p. 258–267.

VIR, P.; MING, S.; TAN, F. Y. **An Empirical Investigation of Code Contribution, Communication Participation, and Release Strategy in Open Source Software Development: A Conditional Hazard Model Approach**. 2007.

WALKER, R. J. et al. Efficient mapping of software system traces to architectural views. In: **Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research**. [S.l.]: IBM Press, 2000. p. 12–.

WEN, Z.; TZERPOS, V. An effectiveness measure for software clustering algorithms. In: **Proceedings of the 12th IEEE International Workshop on Program Comprehension**. Washington, DC, USA: IEEE Computer Society, 2004. (IWPC '04), p. 194–. ISBN 0-7695-2149-5.

WONG, K. et al. Structural redocumentation: a case study. **IEEE Software**, v. 12, n. 1, p. 46–54, Jan 1995. ISSN 0740-7459.

XIAO, L. Quantifying architectural debts. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2015. (ESEC/FSE 2015), p. 1030–1033. ISBN 978-1-4503-3675-8.

YAN, H. et al. Discotect: A system for discovering architectures from running systems. In: **In Proc. 26th International Conference on Software Engineering**. [S.l.: s.n.], 2004. p. 470–479.

ZAIDMAN, A.; DEMEYER, S. Automatic identification of key classes in a software system using webmining techniques. **J. Softw. Maint. Evol.**, v. 20, n. 6, p. 387–417, Nov. 2008. ISSN 1532-060X.

ZIMMERMANN, T.; NAGAPPAN, N. Predicting subsystem failures using dependency graph complexities. In: **Proceedings of the The 18th IEEE International Symposium on Software Reliability**. Washington, DC, USA: IEEE Computer Society, 2007. (ISSRE '07), p. 227–236. ISBN 0-7695-3024-9. Available: <http://dx.doi.org/10.1109/ISSRE.2007.19>.

Appendix

APPENDIX A

Experiment Design

A.1 Experiment with Students

A.1.1 Experiment Design

Next sections are described details concerning to experiments with students.

Institutions 1 and 2 : Students performed comprehension activity and there was the distribution of three groups of subjects: On institutions 1 and 2 we considered three groups to perform comprehension activity: one group performed one activity to evaluate traditional documentation; one group performed activities to evaluate traditional documentation and documentation based on key classes; and the third group performed activities using documentation based on key classes.

A.1.1.1 Experimental Procedure

The experimental procedure was conducted in the classroom by the researcher.

Institutions 1 and 2: Considering **environment setting** for the experiment with students from institutions 1 and 2, there was no need to perform a specific configuration in the execution environment. The only concern was to ensure access to Internet network, since the questionnaires were made available via the web.

Instruction and Training: Before the activities execution, all participating subjects were also instructed, providing materials and training on the experimental procedure in order to reduce failures, deviations and doubts on the performance of the activity. In this way, a minimum knowledge required for the participants is expected, minimizing problems due to lack of knowledge about the use of documentation, the environment used and the execution process. A documentation containing instructions on the procedure was made available for the students. In addition, in the classroom, before beginning the experiments, the researcher reinforced the instructions again. After the initial instructions, we introduce students to documentation structure

based on key class and the traditional documentation of the target applications, highlighting the organization and concepts to assist students during the exploration of documentation. The training lasted 20 minutes, with 10 minutes for each documentation. The activities were described textually on a document to help understanding the problem and then distributed to the students.

Experimental Execution: After the selection and instruction of the students subjects, we defined with the teacher responsible for the class, the days and times to perform out the experiments. In particular, for the two experiments performed with the students, the variable object among the groups during the execution of the experiments was the kind of documentation used (traditional, based on key classes or both), being defined as independent variable of the experiment.

Institutions 1 and 2: Before the experiment execution, for students from institutions 1 and 2, only the documentation of the target application, the questionnaires and the instructions to execute the experiment were available.

In sequence, all students were re-instructed, distributed in groups, the application documentation was briefly presented for each group. Each activity had a maximum duration of 60 minutes. Students were asked to record the start and end time of activities and then complete the questionnaire related to that activity and the group that it belongs. If the student did not complete the activity within the maximum period, the student was instructed to stop the activity and fill out the questionnaire. To fill to the questionnaire the students had 20 minutes to conclude action.

Students could quit at any time if they recognized they could not finish the activity or partially complete it. However, even in these cases, subjects were asked to answerer to one questionnaire.

Institutions 1 and 2 answered one questionnaire, the questions were answered using Likert scale (from -2 to 2).

A.2 Survey with Developers

Next sections are described details concerning to survey with developers.

A.2.1 Survey Design

We conducted a survey that exclusively involved developers of four applications (Apache PDFBox, Service Order, Scholar and Financial). Following, we describe the used methodology:

Procedure: Compared the previous studies, the survey procedure is different. For example, experiments with students were conducted in the classroom by the researcher, whereas the survey was conducted remotely. In this context, the time and environment to evaluate the documentation by the developers were not controlled.

Environment Setting: For the experiment with developers, there was no need to perform a specific configuration in the execution environment. The only concern was to ensure access to Internet network, since the documentation and questionnaires were made available via the web.

Instruction and Training: Before the activities execution, all participating subjects were also instructed, providing materials and training on the experimental procedure in order to reduce failures, deviations and doubts on the performance of the activity. In this way, a minimum knowledge required for the participants is expected, minimizing problems due to lack of knowledge about the use of documentation, the environment used and the execution process. A video containing an example of documentation based on key class was sent to motivate and guarantee a greater adhesion of the developers. Next, emails were sent containing instructions on the steps of the experiment, files and link of the produced documentation.

Experimental Execution: Developers had the flexibility to perform the evaluation of the documentation on convenient days and times, restricted only by a deadline. An email was sent to the developers containing a set of instructions necessary to evaluate the new documentation. They were instructed to evaluate the documentation that was sent as an attachment or made available via web link, and then fill out a questionnaire when the activity was completed. These developers could quit at any time if they recognized they could not finish the activity or partially complete it. However, even in these cases, subjects were asked to answer to the questionnaire. The questionnaire for data collection was adequate for each type of experiment and subject aiming to collect different information regarding the assessed documentation. Throughout the process, the researcher has been available to answer questions. The questions were answered using Likert scale (from -2 to 2). We also consider a set of open questions that will be used to explain the results.