Universidade do Minho
Escola de Engenharia

Informatics Department

Master in Informatics Engineering

Dissertation

Comment Analysis for Program Comprehension

José Luís Figueiredo de Freitas

Supervised by:

Pedro Rangel Henriques, Universidade do Minho
Daniela da Cruz, Universidade do Minho

Braga, October 2011

Abstract

The constant demanding, mostly from Software Maintenance professionals, so that it could be created new and more efficient methods of understanding programs, have put several challenges to Program Comprehension researchers. Nowadays, the programmers are not satisfied with the simple extraction of the program's structure, which mainly resides on the control and dataflow identification. They want to know the meaning of the program, so they can identify the real world concepts that are materialized on the source code of the program, which would provide a better and more efficient understanding of the program.

To solve this problem, Program Comprehension researchers have, mainly, developed approaches that use techniques which are based on Information Retrieval Systems, like search engines. The strategy involves the retrieval of non structured information, properly ranked, answering a question the system can interpret. Considering Program Comprehension, this strategy enables the programmers to search for real world concepts, also known as Problem Domain, implemented and mapped into programming concepts, also known as Program Domain.

Although its use is not consensual, source code comments have the main objective of helping understand the source code, and it has already been proven its value on the process of comprehending, through several studies. Even though the reasons for this fact have not yet been proved, some authors have defended that source code comments are an important vehicle for the inclusion of Problem Domain information and that their exploration improves and increases the comprehension process.

Therefore, the presenting Master Dissertation exposes the development of a solution based on Information Retrieval algorithms, based on Information Retrieval Systems, in order to check if, in fact, the information included in comments can contribute in a decisive way to increase the efficiency of the comprehension process of a given program or software system.

The results and conclusions, extracted from this work, showed that comments, properly analyzed and classified by the developed system, helped to better understand the Problem Domain concepts and their materialization on the source code. The developed solution showed itself able of meeting the challenges posed, and proved to be a usefull and efficient tool for the comprehension tasks that may emerge on the process of software maintence of a system.

Resumo

As constantes exigências, principalmente vindas dos profissionais envolvidos na manutenção de software, para que sejam encontradas novos e mais eficientes métodos para entender programas, têm posto variadíssimos desafios aos investigadores na área de Compreensão de Programas. Atualmente, os programadores não ficam satisfeitos com a simples extração da estrutura do programa, que passa pela identificação principalmente do controlo e do fluxo de dados do programa. Eles querem saber qual a semântica do programa, de maneira a poderem identificar os conceitos do mundo real que estão materializados no código fonte do programa, o que proporcionaria uma melhor e mais eficiente compreensão do programa.

Para resolver este problema, os investigadores da área de Compreensão de Programas têm, maioritariamente, criado abordagens que utilizam técnicas baseadas nos sistemas de Information Retrieval, como os motores de busca. A estratégia passa por retornar informação, não estructurada, devidamente classificada, respondida a uma pergunta que o sistema consegue interpretar. Em relação à Compreensão de Programas, esta estratégia permite aos programadores procurarem os conceitos do mundo real, também conhecido como Domínio do Problema, implementados e mapeados em conceitos da programação, também conhecido como Domínio da Programação.

Apesar da sua utilização não ser consensual, os comentários no código fonte têm o principal objetivo de ajudar a compreender o código, e já foi provado a sua utilidade no processo de compreensão através de vários estudos. Apesar de ainda não ter sido provada o porquê deste facto, alguns autores têm defendido que os comentários são um importante veículo na inclusão de informação do Domínio do Problema do programa e que a sua exploração melhora e aumenta a eficiência do processo de compreensão.

Sendo assim, a presente Dissertação de Mestrado expõe o desenvolvimento de uma solução baseado nos algoritmos de Information Retrieval, baseados nos sistemas de Information Retrieval, de maneira a tentar perceber se, de facto, a informação contida nos comentários conseguem contribuir de forma decisiva para aumentar a eficiência do processo de compreensão de um dado programa ou sistema de software.

Os resultados e conclusões retirados deste trabalho mostraram que os comentários, devidamente analisados e classificados pelo sistema desenvolvido, ajudaram a perceber devidamente os conceitos do Domínio do Problema e as suas materializações no código fonte. A solução criada mostrou-se capaz de responder aos desafios lançados, e provou ser uma ferramenta útil e eficiente para as tarefas de compreensão que possam surgir no processo de manutenção de software.

Acknowledgements

First of all, I would like to thank Professor Pedro Rangel Henriques and Professor Daniela da Cruz for all the help and advices given throughout this year. It would not have been possible to create this work without that essential contribution.

I would also like to thank my parents, António Freitas and Helena Figueiredo, and my brother, João Freitas, for providing all the support needed to overcome this difficult, but important year of my life. The acknowledgments extend to the rest of the family.

Last but not least, the most important acknowledgment goes for my girlfriend, Bruna Ferreira. You always said the right word at the right moment, which gave me strength to carry on, and prevented me to fail or give up. You always understood the importance of this work in my life and supported me every day throughout this year. This dissertation is dedicated to you.

Contents

List of Figures x

List of Tables xi

List of Equations xii

Listings xii

Acronyms xiii

I Introduction 1

1 Introduction 2

1.1 Overview 2
1.2 The Problem 4
1.3 Goals to Achieve 6
1.4 Outline 7

II Program Comprehension 9

2 Program Comprehension 10

2.1 Mental Model 12
2.2 Static elements of the Mental Model 12
2.2.1 Text structure 12
2.2.2 Chunks 13
2.2.3 Plans 13
2.2.4 Hypotheses 13
2.3 Dynamic elements of the Mental Model 14
2.3.1 Chunking 14
2.3.2 Cross-referencing 14
2.4 Cognitive enablers of the Mental Model 15
2.4.1 Beacons 15
2.4.2 Rules of discourse 15
2.5 Knowledge Domains 15
2.5.1 Ontology 15
2.5.2 Problem Domain 16
2.5.3 Program Domain 17
2.6 Cognitive Model 17

2.6.1 Brooks model 18
2.6.2 Soloway and Ehrlich model 18
2.6.3 Shneiderman and Mayer model 18
2.6.4 Pennington model 19
2.6.5 Letovsky model 19
2.6.6 Soloway, Adelson and Ehrlich model 20
2.6.7 von Mayrhauser and Vans model 20
2.7 Program Comprehension tools 20
2.8 Summary 24

3 Information Retrieval for Program Comprehension 26
3.1 Information Retrieval 27
3.1.1 The Information Retrieval Process 28
3.1.2 Vector Space Model 30
3.1.3 Latent Semantic Indexing 30
3.1.4 Probabilistic Latent Semantic Indexing 32
3.1.5 Latent Dirichlet Allocation 33
3.2 Information Retrieval For Program Comprehension 33
3.3 Summary 37

III Comment Analysis for Program Comprehension 39

4 Source Code Comments 40
4.1 Types 41
4.2 Content 42
4.3 Comments and Language 44
4.4 Comments and Program Size 45
4.5 Comment Density and Practice 45
4.6 Summary 46

5 Comment Analysis For Program Comprehension 48
5.1 The role of comments on Program Comprehension 48
5.2 Comment Analysis Program Comprehension tools 50
5.3 Summary 51

IV Darius 52

6 Darius: The first stage 53
6.1 The Preliminary Study 54
6.1.1 Comment Quantity 54
6.1.2 Comment Content 55
6.2 Darius: Version One 56
6.2.1 Extracting and Locating Comments 57
6.2.2 Comment Statistics 59
6.2.3 Comment Words Analyzer 60
6.2.4 Graphical Interface 61

6.3 Tests: Preliminary Study 64
 6.3.1 Objects of Study 66
 6.3.2 Comment Quantity 66
 6.3.3 Comment Content 70
 6.3.4 Discussion of the results 76
6.4 Summary 77

7 Darius: The second stage **79**
7.1 Darius: Version Two 80
 7.1.1 Document Database or Corpus Implementation 81
 7.1.2 Vector Space Model Implementation 82
 7.1.3 Latent Semantic Analysis Implementation 83
 7.1.4 Graphical Interface 84
7.2 Test 86
 7.2.1 iText 87
 7.2.2 Concept Location 87
 7.2.3 Discussion of the results 96
7.3 Summary 98

V Conclusion **100**

8 Conclusion **101**
8.1 Discussion and Conclusions 103
8.2 Relevant work 105
8.3 Future work 105

Bibliography **106**

List of Figures

2.1 Example of an ontology of the Problem Domain 17
2.2 Codecrawler's Class Blueprint view, extracted from [49] 22
2.3 JRipples interface, extracted from [19] 23
2.4 SHriMP view of a graph node, extracted from [65] 24

3.1 Information Retrieval system main process 28

6.1 Darius First Version Structure 57
6.2 Darius Graphical Interface 62
6.3 Button to load a Project/Program 62
6.4 Loading a Project/Program 63
6.5 Comments Statistics Calculator Graphical Interface 63
6.6 Results from the Comments Statistics Calculator 64
6.7 Comments Words Analyzer Graphical Interface 65
6.8 List of words to analyze 65
6.9 Results from the Comments Words Analyzer 66

7.1 Darius Second Version Structure 81
7.2 Concept Location Graphical Interface 85
7.3 Loading an IR model 85
7.4 Writing a query to execute 86
7.5 Results from a query 86
7.6 iText Problem domain 88
7.7 PDF Document query 89
7.8 PDF Writer class query 89
7.9 Document's title query 90
7.10 Add title to document method 90
7.11 Meta query 91
7.12 Anchor query 92
7.13 Phrase query 92
7.14 Chunk query 92
7.15 Paragraph query 93
7.16 Section query 93
7.17 Chapter query 94
7.18 Image query 94
7.19 List query 95
7.20 List Item query 95
7.21 UML Class Diagram of the results 96

List of Tables

6.1 Description and size of each selected project 67
6.2 Comments Frequency in the projects (detailed analysis). 68
6.3 Percentage of Source Code Entities (SC) commented (#SC commented / #SC) . . 69
6.4 Most used type of comment per type of source code entity 69
6.5 Percentage of domain words (DW) found (#DW Found / #DW) 71
6.6 Frequency (%) of words of each Domain (#Total DW / #Total Words) 72
6.7 Frequency (%) of words of each Domain per type of comment 72
6.8 Percentage of each type of comments that contains the type of Domain words . . 72
6.9 Frequency (%) of words of each Domain per source code entity 74
6.10 Percentage of each source code entity comment that contains the type of Domain words 75

7.1 Results from the Queries 97

List of Equations

3.1 Term Frequency - Inverse Document Frequency 29
3.2 Bag-of-words 30
3.3 Cosine Similarity (VSM) 30
3.4 Single Value Decomposition 31
3.5 Probabilistic Latent Semantic Indexing 32
3.6 Latent Dirichlet Allocation 33
3.7 Cosine Similarity (LDA) 33
7.1 Cosine Similarity (LSA) 84

Listings

4.1 Inline Comment 42
4.2 Block Comment 42
4.3 JavaDoc Comment 42
4.4 Unit Comment 43
4.5 Range Comment 43

Acronyms

AST	Abstract Syntax Tree. 58
FCA	Formal Concept Analysis. 36, 37
IR	Information Retrieval. 21, 27–30, 33–37, 40, 48, 50, 51, 53, 79–84, 86, 88, 97, 98, 101–103, 105
LDA	Latent Dirichlet Allocation. 33, 37
LSA	Latent Semantic Analysis. 30–32, 34–37, 83, 85, 97, 98, 103, 105
LSI	Latent Semantic Indexing. 30
PC	Program Comprehension. 3–8, 10–12, 17, 20, 21, 23--27, 33–37, 40, 48–51, 53, 54, 64, 67, 70, 76–80, 84, 86, 87, 90, 98, 101–104
PLSI	Probabilistic Latent Semantic Indexing. 32, 33, 37
SVD	Single Value Decomposition. 31, 32, 83
TF-IDF	Term Frequency - Inverse Document Frequency. 29, 82
UML	Unified Modeling Language. 95, 96
VSM	Vector Space Model. 30, 37, 82–85, 97–99

Part I

Introduction

Chapter 1

Introduction

This document is a master dissertation that takes part of the second year of the Master Degree in Informatics Engineering that is held at University of Minho in Braga, Portugal. The work presented in this master dissertation is included in the area of Program Comprehension and Comment Analysis.

1.1 Overview

We live, without any questions, in the age of Information Technologies. Our lives are increasingly being more dependent on Information systems, that are addressing our old needs and creating new and different necessities. The society's continuous requirement for the improvement of these systems, as taken the professionals of the area to think about how to develop Information systems faster and cheaper. In software, the professionals of Software Engineering have created, along the years, a large number of processes and tools that respond to those requirements. Nowadays there is almost an infinite number of methods and approaches that software engineers and programmers can follow and use to develop their product according to its requirements, to integrate more easily on project management and to reduce time and financial costs.

In theory, every process chosen for the development of a software product has the main objective of a creating a flawless artifact with all the projected requirements developed. But the reality is that almost every software product does not contain the total set of the planned requirements, and in those which were developed, some are inefficient or contain severe errors, known as bugs. Software Maintenance, has become, then, one of the most resource spending phase of every software development model. According to [25], before the early 1990s, the maintenance of a software

product cost almost half of the resources given for its development, and recently in [32] it was mentioned that on industry, 80 to 95% of the budget given to Information Systems' development is spent on maintenance activities.

In Software Maintenance, the biggest problem still resides on the process of comprehending the program or system to maintain. In [35] the author stated that half of the time spent in Software Maintenance, is for trying to understand the program to maintain.

The task of comprehending a program is not always an easy and straightforward task. There are a number of factors that can aggravate this process: the large size and high number of source code files, the poor quality of the source code, an unknown or not familiar programming language used, the poor quality of identifiers and comments, amongst others. The combination of some or all of these factors can almost make impossible the task of understanding a program.

Like any other cognitive process, understanding an unfamiliar or forgotten program has its own particular characteristics. Every person who tries to understand a program, uses a different cognitive process by extracting information from different source code elements. The diversity in cognition has been subject of several studies, particular from the researchers of *Program Comprehension (PC)*. **PC** professionals have the purpose of studying the mental strategies that programmers follow in order to understand a program. By doing this, these professionals can develop tools (**PC** tools) that can catalyze the comprehension process. Considering that most of the time in manual program comprehension is spent on source code reading, these tools contribute enormously for a faster comprehension process.

Most of these tools enhance program comprehension through the analysis of the source code itself, using techniques from Program Slicing, control-flow analysis or execution trace analysis, amongst others. Considering these techniques, it can be concluded that they can extract the structural information from source code, helping the programmer to understand how the program executes, which and in what order the functions are called, which variables are used in one particular moment, and other similar information. However, all this information is not sufficient to really understand a program.

Every program is contextualized in a domain, containing real world concepts, that is emulated on the source code. Every instruction and variable has its own meaning that contributes for a better approximation to the real word. The semantic of source code is what connects the programmer to the program, and in this way is what contributes most for program comprehension.

Thinking about the semantic in source code, comments are probably the most practical form of expressing it. Comments have the enormous power of explaining source code along it, in a natural language that programmers understand. Good comments can contain the explanation of source code instructions connecting the semantic to the structure. In a perfect world, it could be said that comments is the interface between two languages: the natural language and the programming language.

Considering the important semantic richness of source comments it becomes essential to use it to enhance **PC** tools. A tool that extracts information from comments to help programmers understand a program, turns into a powerful instrument for **PC**.

Some steps were taken in order to develop **PC** tools that are based on comment information. However, there is still much to do and the capacity of comment information was not explored at its most. It becomes, so, one of the main goals for **PC** researchers.

1.2 The Problem

The role of comments in **PC** has been subject of several studies along the years. *Brooks* referred in his work about **PC** [17], some words from a late study from *Kernighan and Plauger* about good programming styles [43], which concluded that the best documentation for a program includes enlightening comments. The theory presented in this work has also defined the role of comments on programming, towards a better program understanding. According to *Kernighan and Plauger*, comments should contain the code explanation by creating a bridge between the code and the operations on real world concepts and objects that are materialized in the code.

Corroborated by *Brooks*, the bridge which connects the Problem Domain and Program Domain has been accepted, amongst several cognitive models, as a decisive process in the comprehension of a program.

Though comments can themselves contribute for the comprehension process, the construction of the domains bridge by only reading all the comments, represents a long task. So, if it could be automatized this process by retrieving the information in a strategic way to the user, the process of comprehension a program will be faster. It becomes essential, then, to take advantage of comments to create a **PC** tool.

Accepting that some steps were taken in that direction, the potential of comment information has

not been explored to the fullest. Specially, if taking in consideration that the power of comments reside in creating the domains bridge, **PC** tools have not addressed the problem exploiting that capacity.

However, stepping in that direction represents a difficult task, that can explain the lack of Comment Analysis **PC** tools. The lack of this kind of tools can be due to two factors: the lack or small number of comments on the analyzed program, or/and their poor quality. If a program does not contain enough comments, or if it contains, their quality is questionable, a Comment Analysis **PC** tool will be inefficient and useless.

So the development of a Comment Analysis **PC** tool should take in consideration all the mentioned aspects. And the first question that arises from these conclusions is the following:

Question 1. *Is it worthwhile to use comments to develop a Comment Analysis PC tool?*

A positive answer to this question is dependable on the positive answers on the following two questions:

Question 2. *Do programmers usually include a significant number of comments on their source code?*

Question 3. *Do comments usually include information regarding the problem and program domain?*

Question 2 addresses the problem of the lack of comments in the source code. A positive answer to this question is crucial to answer positively to Question 1 because if programs do not contain comments or have a small number of comment percentage, it will be useless to develop a Comment Analysis **PC** tool. It is important to refer that to answer this and Question 3, it is not necessary to study every program that exists in the world, which would be obviously impossible. If it is proven that in a significant set of programs, exists a sufficient number of comments, it can be given a positive answer to Question 2.

About Question 3, it addresses the problem of the lack of quality of comments content. As mentioned before, *Kernighan and Plauger* defined that enlightening comments should contain information to create the domains bridge. In order to create this bridge, comments must contain information from the two domains: Problem and Program Domain. So it can be defined as a sign

of quality on the comments content, the use of Problem and Program Domain information. If comments contain these two kinds of information to create the domains bridge, they can be very effective on **PC**.

Answering positively to Questions 2 and 3 will prove the feasibility of creating a Comment Analysis **PC** tool. However the simple fact that it is feasible to create this tool, does not prove that it will be efficient to enhance the comprehension of programs. So the following question arises:

Question 4. *Can the information regarding the Problem and Program Domain contained on comments, properly retrieved and analyzed by a Comment Analysis PC tool, be used to enhance the program comprehension by establishing a bridge between the two domains?*

At first glance, and taking in consideration *Kernighan and Plauger*'s theory, it could be answered positively to this question. If comments are a privileged mean to involve Problem and Problem Domain information, Comment Analysis can be used to create a **PC** tool. However, as mentioned before, there is still a lack of Comment Analysis **PC** tools, so we can assume the difficulty on using comments to develop such tools. So, in order to do so, some effort will have to be spent, in finding the best strategies to take advantage of comment information to enhance **PC**, by presenting the information in a certain way so that users can create faster the domains bridge.

1.3 Goals to Achieve

In the last two sections, it was discussed the need for a Comment Analysis **PC** tool. As mentioned before, the exploration of comments through a **PC** tool would create a great impact on **PC**. Taking this in consideration, the work presented in this document had the main objective of studying and creating an approach that analyzes comments in order to improve program understanding, which was implemented on a Comment Analysis **PC** tool.

However, as referred on the previous section, the feasibility of developing such a tool must be studied in detail; in order to do so, Question 1 must be properly answered by addressing Questions 2 and 3. If the feasibility is proven, the development of an approach can be carried out.

Taking in consideration all the mentioned aspects, the goals of this work were the following:

- Study the fundamental definitions, cognitive models, approaches and tools of **PC**;
- Study the role of comments on **PC**;

- Study techniques which can be applied to develop a Comment Analysis for **PC** tool;
- Prove the feasibility of developing a Comment Analysis **PC** tool;
- Develop a Comment Analysis **PC** tool prototype, using the studied techniques and taking in consideration the role of comments on **PC**;
- Evaluate the developed prototype through a rigorous test.

Considering the goals for this work, it is safe to state that all of them were fulfilled with success. All the results and conclusions, as the goals were accomplished, are described throughout this document.

1.4 Outline

In this section it is given an overview and a resume of all the chapters presented on this dissertation.

In Chapter 2, it is introduced the area of Program Comprehension, which is a major topic for this dissertation. It starts with some fundamental definitions, which include the reference of mental model and its characteristics, mainly its static and dynamic elements and its cognitive enablers. One important topic included in this chapter is the notion of knowledge domain. This topic is fully detailed, and it is given focus for two examples of knowledge domain: the Problem and the Program Domain. With the fundamental definitions fully detailed, the chapter continues with the enumeration and description of the main cognitive models, accepted on the literature. The chapter ends with an enumeration of the most important **PC** tools referenced on the literature.

In Chapter 3, it is given an overview of the role of Information Retrieval on the Program Comprehension area. To fully understand this role, it is introduced a detailed explanation of the Information Retrieval area, including the exposition of the main process executed on Information Retrieval and the enumeration of the main models used on Information Retrieval. The chapter ends with a study of the current state of the approaches created in order to incorporate Information Retrieval techniques to enhance **PC**.

In Chapter 4, it is included a detailed study of the source code comment, which includes an overview of the available literature. This includes the enumeration and characterization of the different types of comments and comments content. It is also given an outlook of the relationships

between the source code comment and the natural language and with the size of the program. Concluding this study, there is included a description of the perspective on the literature of comment density and the practice on software development in the end of this chapter.

In Chapter 5, it is introduced a main topic of this dissertation which is the area of Comment Analysis for Program Comprehension. This chapter is divided into two different sections that addressed two different subjects of this area. The first focuses on the overview on the literature of the role and function of the comment as an important tool for understanding programs. It includes a detailed enumeration of several tests with human subjects that prove that important role. The second section focuses on the enumeration of approaches that use comment analysis to attain **PC**, and that give the current state of the area of Comment Analysis on the Program Comprehension field.

In Chapter 6, the focus is on providing answers to Question 1 raised in this dissertation. It includes the development of a preliminary study that tried to respond to that challenge. The goals and strategies of this preliminary study are fully detailed in this chapter. Then it is described the first version of **Darius**, a tool developed in this work, which is able to run that preliminary test. In the end of this chapter, it is included the description of the steps taken to develop this study, and a discussion of the obtained results.

In Chapter 7, the main concern is the strategies developed in order to provide a positive answer to Question 4. It includes the total detailed description of every module and from the graphical interface, of the second version of **Darius**, where it is able to locate problem domain concepts on the source code, using comment information. This chapter concludes with the enumeration and description of the steps taken to execute a test that explored the potential of **Darius**. The presentation and discussion of the obtained results are also included in this chapter.

This dissertation is concluded by Chapter 8, where there is presented the main conclusions extracted from this work, and its contributions to the field. There is also pointed out some further topics of investigation, extracted from the conclusions of this work.

Part II

Program Comprehension

Chapter 2

Program Comprehension

Software Maintenance is defined as the *modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to a changed environment* [24]. Knowing the important role of *Software Maintenance* on software development, the fact that it spends so many resources mainly on the comprehension of the system, as previously mentioned, have created several challenges for researchers. They have identified five tasks that demand the comprehension of a program in *Software Maintenance* [100, 23, 9]:

- Adaptive: adapting software to a new data and processing environment;
- Perfective: improving performance, processing efficiency or maintainability;
- Corrective: correct software in terms of implementation failures, processing or performance;
- Reuse: developing a new version of a system by using some of its old components;
- Code leverage: developing a new version by modifying the old system.

Bearing in mind the description of these tasks, it can be stated that *program comprehension is a vital blend of software engineering activities that supports reuse, inspection, maintenance, evolution, migration, reverse engineering, and reengineering of existing software systems* [45]. Recognizing the vital role of program comprehension as an activity, researchers have studied along the years the way programmers comprehend a program [17, 87, 72], creating in this manner the discipline of **PC**, which is defined in [12] as a *Software Engineering discipline which aims to understand computer code written in a high-level programming language*.

In fact, according to **PC** researchers, in order to comprehend a program, programmers do not have the same cognitive mechanisms. They follow different cognition strategies according to different factors [84]. These factors were identified in [92], and can be aggregated into three groups: *Maintainer*, *Program* and *Task factors*. The first group is related with the characteristics of the programmer or the individual who is trying to understand the program. These include the Program Domain knowledge that he has, and his expertise, experience, creativity or familiarity with the program. The second group aggregates factors from the program itself, such as its Program Domain, its size, complexity and quality, and the availability of its documentation. The final group, aggregates factors related with the comprehension task at hand, which includes the task's type, size and complexity, time constraints and environmental factors. The variation of one or some of these factor can contribute for a change in the chosen strategy.

Regardless of the strategy followed, comprehending a program does not always represent an easy task. According to [81], the main problems in program understandability are due to the difficulty that programmers have in establishing bridges between different conceptual areas, such as between:

- the Problem or Application domain and the solution;
- different types of abstraction levels in programming;
- the description or design of the system and the actual developed one;
- the associational nature of human cognition and the formal world of software;
- the analysis of the source code using a bottom-up approach and the synthesis of the application's description using a top-down approach.

Recognizing the difficulties that programmers have in comprehending a program, as well as the different strategies that they follow to do it, **PC** researchers have studied these characteristics in detail, providing different explanations for the cognitive processes in the comprehension of a program. As the field of **PC** becomes richer, the process of comprehending a program has become a little more clear.

In the following sections, the main topics of the **PC** research are addressed. In Section 2.1 is it introduced the topic of *Mental Model* by giving a definition of it. This topic is continued on

Section 2.2, where there are enumerated and detailed the static elements of the mental model, and on Section 2.3 there are enumerate the dynamic elements. This topic is concluded on Section 2.4, with the enumeration of the cognitive enablers of the mental model. On Section 2.5 it is introduced an important topic of the **PC** research, which is the concept of *knowledge domain*. Apart from its definition, this section also focuses on the analysis and definition of the Problem and of the Program Domain. On Section 2.6 there is given a definition of *Cognitive Model*, and it is also included the most referenced cognitive models available on the literature. This chapter ends with Section 2.7, with a list and description of tools that enhance **PC**.

2.1 Mental Model

In [92], *Storey et al.* defined *Mental Model* as *a maintainer's mental representation of the program to be understood.* As a mental representation, it is formed through observation, inference or interaction with the program [84]. This process of formation can be more or less correct or complete according to several factors such as the programmer's experience or the program complexity, as mentioned before.

According to *von Mayrhauser and Vans*, a mental model is composed by static and dynamic elements [100], and its construction is catalyzed by cognitive enablers. In the following sections, these elements are addressed.

2.2 Static elements of the Mental Model

Although the construction of a mental model is different from programmer to programmer, every mental model contains a set of entities which are the same on all cases. The static elements that constitute a mental model include *text structures, chunks, plans and hypotheses*. These are described in detail in the following subsections.

2.2.1 Text structure

According to *von Mayrhauser and Vans* in [100], *a text structure includes the program text and its structure*. In more detail, the text structure is the enumeration, the sorting and the ranking of instructions that constitutes the organization of a program [11].

2.2.2 Chunks

Chunks [51, 26] are text structures which represent different levels of abstraction. In some cases, a chunk can be formed by several other chunks with lower levels of abstraction.

A single chunk is composed by a *microstructure* and a *macrostructure*. A chunk's *microstructure* constitutes its set of statements and instructions, which can be abstracted by its *macrostructure*, which includes only a *label* that describes the chunk. For example, a chunk which constitutes a text structure of the sort algorithm, can be attributed the label *sort*.

2.2.3 Plans

In [100], *plans* are defined as *knowledge elements for developing and validating expectations, interpretations, and inferences*. In other words, a plan is a stereotyped situation that programmers can use in order to infer or recognize that typical situation from the program. They are composed by *types* and *fillers*. The first ones are generic objects such as *lists*, *arrays* or *stacks* and the second ones are specific procedures and operations on those objects, such as *pop* and *push* for a stack object. These are examples of *programming plans*, as their domain includes programming concepts. The other type of plan is the *domain plan* and is related with the problem or application area of the program. A typical *domain plan* of a music store program would include objects such as a *CD* or a *DVD* and operations such as *buy a CD* or *create a DVD*.

2.2.4 Hypotheses

Hypotheses [18] are conjectures or plausible inferences that the programmer formulates about the program. In [52], *Letovsky* identified three major categories of hypotheses:

- *Why hypotheses*. Conjectures about the purpose of a program entity;
- *How hypotheses*. Conjectures about the way a goal is accomplished in the program;
- *What hypotheses*. Conjectures about the classification of a program entity (variable, function, etc.).

2.3 Dynamic elements of the Mental Model

As mentioned before, when trying to understand a program, programmers follow different ways of achieving that goal, according to several factors. The sequence of actions and procedures that programmers follow in order to do it, is defined as *strategy* [100, 54].

There are two different categories of strategies [100], according to the *amount* of the program to understand and to the *detail* of the comprehension analysis. For the first category, *Littman et al.* identified in [54] two different types of strategies that programmers follow: *systematic* and *as-needed*. In the first type, the programmer tries to understand the whole program before any maintenance activity, as opposed to the second one, where the programmer tries to understand only the parts of the program which are targeted for maintenance. In the second category, two types of strategies are identified: *shallow* [87, 1, 20] and *deep reasoning* [20]. *Shallow reasoning* was introduced by *Soloway and Ehrlich* in [87], as a strategy followed by programmers, where they scan for critical lines of code and *beacons* to instantly identify *plans* in their mental model [1]. In *deep reasoning*, the programmer tries to synthesize the program, by analyzing the whole program, and tries to establish causal connections between the functions and objects of the program [100]. This type of strategy is most used when the program is completely unknown to the programmer, as referred in [27].

As a procedure of production of information, a strategy is guided by two processes: *chunking* and *cross-referencing*.

2.3.1 Chunking

Chunking [51, 26] is the process of joining and aggregating several lower level chunks who are conceptually connected to form a higher level chunk, who can then be attributed a label which abstracts the chunk.

2.3.2 Cross-referencing

Cross-referencing [100] is considered to be an essential process on the construction of a mental model [73, 2]. In this process, the programmer tries to connect different levels of abstraction, such as between program parts and functional descriptions [100].

2.4 Cognitive enablers of the Mental Model

Cognitive enablers, as the word says, are elements which can increase the pace of the comprehension process. The following two subsection present two examples of cognitive enablers.

2.4.1 Beacons

Beacons [18, 103] are clues on the program that could give indicators of a particular structure or operation. For example, a block of code which represents a *swapping* operation in an array, can be a beacon for the implementation of a *sort* algorithm.

2.4.2 Rules of discourse

In [87], *Soloway and Ehrlich* defined *rules of discourse* as *rules that specify the conventions in programming.* These include standard algorithms, data structure implementation and coding standards [21].

2.5 Knowledge Domains

The concept of *knowledge domain* was defined by *Brooks* in [18], as *a closed set of primitive objects, relations among objects, and operators which manipulate these properties or relations.* According to this author the comprehension of a program is achieved through the bridging and the mapping between several different abstraction level knowledge domains, such as between the *Problem Domain* and the *Program Domain* [102]. In this section there is included an exhaustive study about this subject subject.

2.5.1 Ontology

Although the representation of a knowledge domain can be purely mental, there are other ways of describing it. One of them is by using an *ontology* [22, 69, 42]. which is *a formal explicit description of concepts in a domain of discourse (classes (sometimes called concepts)), properties of each concept describing various features and attributes of the concept (slots (sometimes called roles or properties)), and restrictions on slots (facets (sometimes called role restrictions))* [69]. A

class is the collection of objects of the *domain* that share *properties*. A class can have *subclasses*, that have more properties and are more specific, and this relationship is defined by a taxonomy. Classes can also be instantiated, creating a *knowledge base*.

The development and use of ontologies have several advantages. These include the sharing and reusing of knowledge, as well as a better acquisition, verification and maintenance of knowledge base systems.

There are several methodologies for the development of ontologies, that can be seen in [42], but in general the following steps are generic [69]:

- Identifying the classes of the ontology;
- Arranging the classes in a taxonomy hierarchy;
- Identifying the slots and the permitted values for them;
- Instantiating the classes.

An example of ontology can be seen on Figure 2.1.

2.5.2 Problem Domain

Problem or *Application Domain* [85, 17] of a program is a part of the world about which the program is concerned with solving problems [82]. It can be characterized by a common shared vocabulary, by common assumptions and approaches for the solutions, and a literature, that exists independently from the programs that solve the problems. In [77], *Prieto-Diaz and Arango* enumerated the prerequisites for a *Problem Domain*:

- deep and comprehensive relationships among the objects of the knowledge domain;
- the existence of a community concerned about solving problems of the domain;
- the acknowledgment in the community of the existance of a software solution for these problems;
- the existence and access to a body of knowledge of the domain to create that solution.

Figure 2.1 shows an example of a Problem Domain of software connected with library management, represented by an ontology.

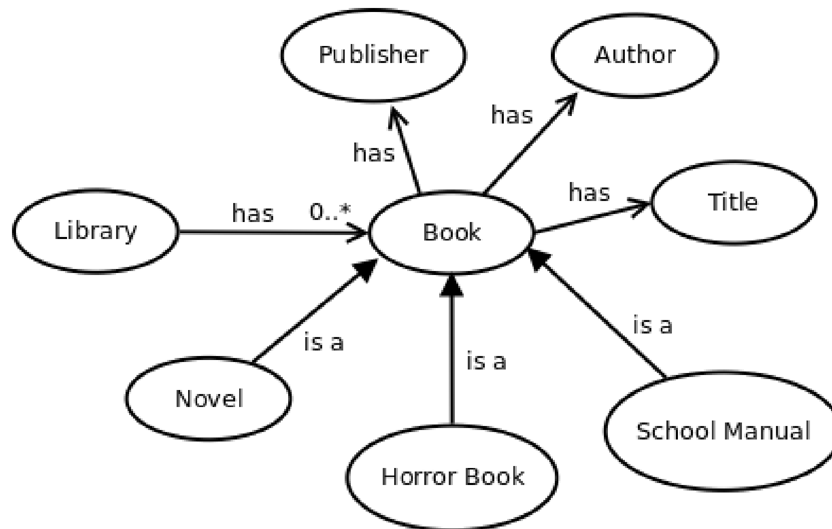


Figure 2.1: Example of an ontology of the Problem Domain

2.5.3 Program Domain

Program or *Programming Domain* [17] of a program is a knowledge domain related with the programming and with the source code level. It includes generic concepts of *programming* and also specific concepts relating to the *programming language* and its *paradigm* (object-oriented, imperative, functional, etc).

2.6 Cognitive Model

In the previous sections it is discussed the concepts behind the mental model, the entities that form it (static and dynamic) and that enable and accelerate its construction, as well as the definition of knowledge domain, highlighting the Problem and Program domain. In this section, it is discussed the concept of *cognitive model* in **PC**. It is defined as a mental pattern that programmers follow in order to formalize the knowledge about a program through the creation of a mental model [101]. This concept of cognitive model is different from the *strategy* one, in the way that the first is defined as the set of *cognitive processes and temporary information structure in the programmer's head* [93], and the second is more defined by the goals of the comprehension and is more rational.

In this chapter it was also enumerated the factors (maintainer, program and task) that contribute to the different types of cognitive processes of programmers when they understand programs. Taking this, researchers in **PC** have created several theories that try to explain these cognitive processes.

In the following subsections it will be described the most relevant cognitive models accepted in the literature. The comparison between models and its discussion can be seen in [11, 100].

2.6.1 Brooks model

The main idea in *Brooks* model [18] is that the comprehension of a program is the result of the mapping between the Problem Domain and the Program Domain. To do so, the programmer has to execute successive mappings over several other domains in between. This paradigm of construction and reconstruction of knowledge is directly dependent on the programmer's expertise about the Problem Domain of the program. Starting with this Domain knowledge, he defines initial hypotheses and then successively refines them using a *top-down* approach through the domains. The refinement process includes the validation of these hypotheses, which can be done, mainly, through the existence of beacons on the source code.

2.6.2 Soloway and Ehrlich model

In [87], *Soloway and Ehrlich* studied the effect of the rules of discourse and plans on the comprehension of a program by experients and novices programmers. This was done in order to support their theory of comprehension. According to the authors, programmers expect that a program is composed of plans which have been adapted to the Problem Domain of the program [105], and that comprehension is dependent on whether the rules of discourse are followed. The results of the study corroborated their theory. As expert programmers have more rules of discourse experience they were faster in the process of comprehension. However, when dealing with unplan-like programs (rules of discourse violated), the times of comprehension were similar to those of novice programmers. This proved in fact their theory.

2.6.3 Shneiderman and Mayer model

The *Shneiderman and Mayer*'s model [86] sees program comprehension as a *bottom-up* task. In bottom-up comprehension, the theory is that programmers start by reading individual code statements and try to successively chunk them into high level structures, *until some high-level understanding of the program is attained* [93]. As for Shneiderman and Mayer's model in particular, the authors considered that comprehension is composed by *syntactic* and *semantic programming*

knowledge. *Syntactic knowledge* is composed by syntactic details of the programming language in particular such as *keywords, language syntax and available library routines* [105]. *Semantic knowledge* includes language-independent programming knowledge and Problem Domain information. Using a bottom-up approach, the programmer tries to reconstruct the Problem Domain of the program, by using these two types of knowledge.

2.6.4 Pennington model

Pennington's model [72] also sees comprehension as a *bottom-up* task. In their model, in program comprehension the programmer defines two mental structures: the *program* and the *situation model*. The program model is build first and extracts the sequence of operations and procedures of the program, defining an abstract control-flow of the program. This is achieved through the chunking of microstructures and cross-referencing into high level programming plans and macrostructures. The situation model is also constructed in bottom-up, through chunking, however, these are done by mapping the program model into high level domain plans (Problem Domain) that describe the goals of the program.

2.6.5 Letovsky model

In *Letovsky*'s model [52], comprehension is defined as a set composed by three elements: a *knowledge base*, a *mental model* and an *assimilation process*. The *knowledge base* is dependent from the programmer and contains information regarding the Problem Domain, the Program Domain, rules of discourse and plans. The *mental model* is composed by three layers. The first one is called *specification layer* which contains the Problem Domain and high level goals of the program. The second layer, *implementation layer*, is composed by the Program Domain of the program. The third layer contains the current established links between the other two layers performed by the programmer, which is called *annotation layer*. In order to create these layers, the programmer follows an *assimilation process*. According to *Letovsky*, this is achieved either by top-down or bottom-up, depending on the way programmers think is the best source of knowledge in the moment.

2.6.6 Soloway, Adelson and Ehrlich model

The *Soloway et al.*'s model [88] defines comprehension as a top-down task which connects *external representations* (documents,requirements, etc.), *internal representations* (mental model), *rules of discourse* and *plans*. In particular, this model highlights the role of plans in comprehension, defining three types of them: *strategic*, *tactical* and *implementation plans*. As this model follows a top-down approach, the programmer starts with an overall idea of the goals of the program, considering the Problem Domain, and according to the external representations and his expertise. The *strategic plan* defines the global strategy or algorithm expected by the programmer to obtain a particular goal in the program. The second plan, *tactical plan*, describes the general steps of the particular algorithm that is chosen to solve the problem. The *implementation plan* defines the expected data structures, functions and operations that implement those steps, and that map them to the source code. In general, this cognitive model defines comprehension as the successive mapping between these three types of plans using, mainly, rules of discourse and beacons.

2.6.7 von Mayrhauser and Vans model

von Mayrhauser and Vans theorized a model described in [99] that is based on the fact that programmers change between top-down and bottom-up, according to the necessities. In particular this model incorporates the top-down model of *Pennington* and the knowledge base model of *Letovsky*. As mentioned before, the knowledge base is used to store knowledge in order to achieve new knowledge. When the program is known and beacons can be recognized, the programmer tends to use the top-down. Then, if the code is unfamiliar to the programmer, he tries to invocate the *program model*, obtaining a control-flow abstraction of the program. When some or all of the program model is built, the programmer tries to map it into higher level structures, defining the *situation model*, as mentioned before.

2.7 Program Comprehension tools

As was mentioned before, the task of comprehending a program is not always easy and straighforward. In order to help in that task, researchers have created several *PC tools*, considering the main topics behing **PC**, that are addressed in this chapter. The main paradigm of these tools is to *extract*

information, analyze and focus it, and finally, visualize it for the person who tries to understand the software system [55]. To do so, a **PC** tool must contain the following modules, as was defined in [12]:

- a parser and an analyzer for the source code of the given program;
- a knowledge base to store and handle the extracted knowledge;
- interactive vizualizers, to present the retrieved information to the user.

In this section some **PC** tools are briefly described. Although **Information Retrieval (IR)** and Comment Analysis **PC** tools do no take part of this list, they are fully detailed in the following chapters as they are of great interest for this work.

Alma

Alma [74] is a tool that provides to the user the inspection of the data and control flow of a program, through vizualization and animation. This particular tool has the advantage of being language independent.

Bauhaus

Bauhaus [79] is a toolkit that provides methods, tools and techniques for **PC** on all levels of abstraction, from source code level to architecure level. In particular, the tools provide architectural extraction and validation, as well as protocol analysis through control flow graphs.

Canto

Canto [5] is a **PC** and Software Maintenance environment that analyzes *C* programs, and extracts the arquitectural structure and the control-flow graph.

Codecrawler

Codecrawler [49] is a Software Vizualization tool that provides views about source code entities and the relationships between them. For **PC** purposes in particular, this tool offers an interesting view called *Class Blueprint* (Figure 2.2), that provides the vizualization of the internal structure of classes

and class hierarchies. In this view, it can be seen, for example, the number of times a method is called by a class or the similarity amongst classes.

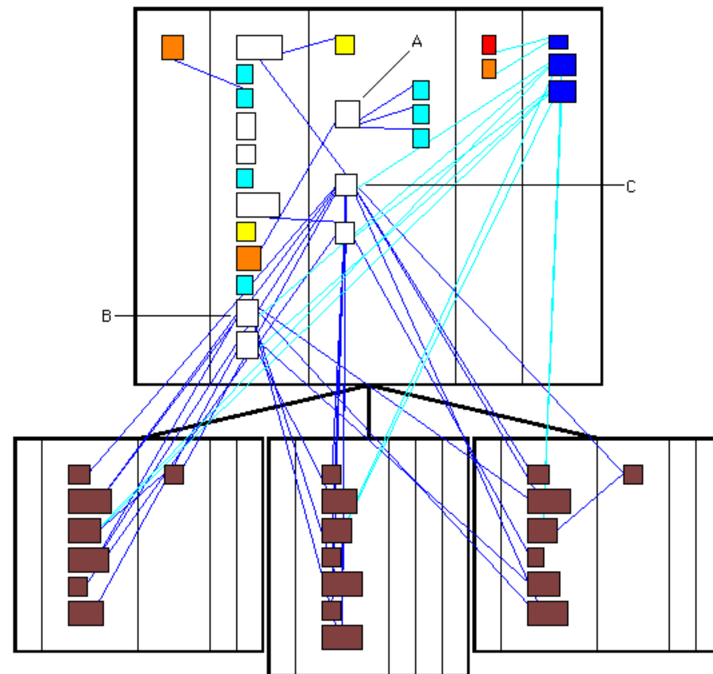


Figure 2.2: Codecrawler's Class Blueprint view, extracted from [49]

Imagix 4D

Imagix 4D[1] analyzes C++ and Java programs, and provides views of the architecure structure at different levels of abstraction. These go from class and function dependencies to data-flow analysis and detailing of data structures and usage.

JRipples

JRipples [19] is a tool that assists programmers on incremental software changes, providing impact analysis and change propagation. The user defines the set of classes which are going to be changed (impacted), and the tool provides the enumeration of classes which are dependent on those, and that need do be changed also, saving time to the programmer, that does not need to comprehend the whole system to see what classes successively need change and are impacted. The interface of JRipples can be seen on Figure 2.3.

[1] http://www.imagix.com/

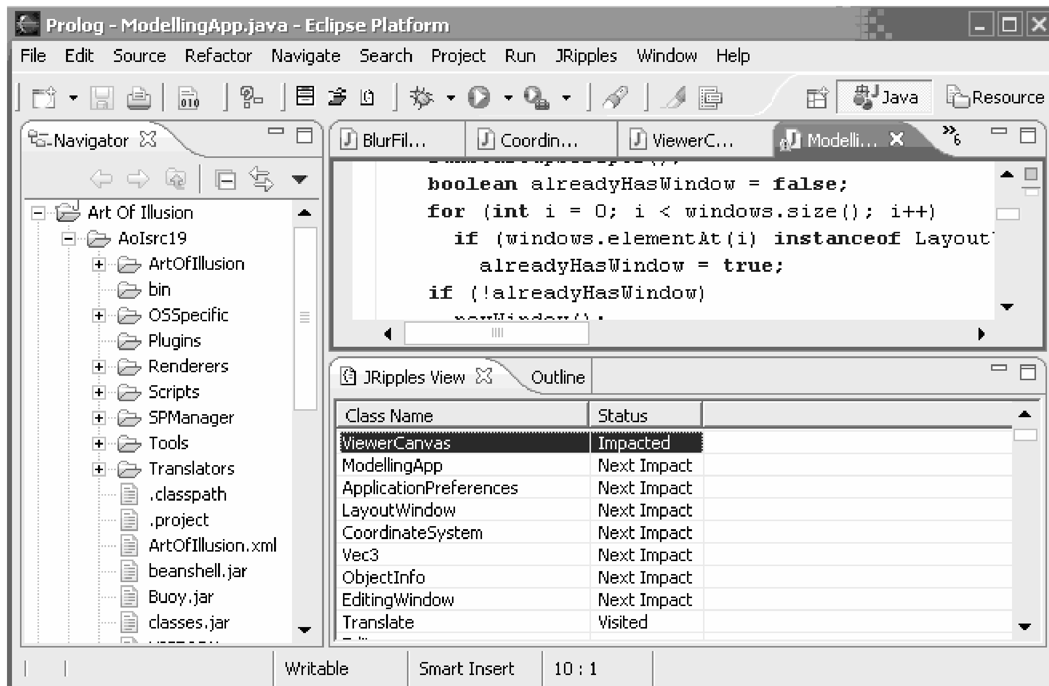


Figure 2.3: JRipples interface, extracted from [19]

Oo!Care

Oo!Care [53] is a **PC** tool developed for object oriented programs. Through *Oo!Care*, the user can extract and vizualize the program dependencies in a control-flow and data-flow graph, the inherence-hierarchy between classes and file-dependencies.

Rigi

Rigi [68] is a **PC** tool for *C*, *C++* and *COBOL* programs. It provides support for the extraction, analysis and vizualization of these software systems. The main idea of *Rigi* is to parse the source code and translate it to an intermediate language (Rigi Standard Format), that can be read by the graph editor. This graph editor is capable of analyzing the information and providing the vizualization of a graph, that can show the architectural structure of the software system at different levels of abstraction. The nodes of the graph include functions, variables or data structures, and the links can represent function to function calls or function to variable calls.

SHriMP

SHriMP [65] is another Software Vizualization **PC** tool that presents a nested graph view of a software architecure, presenting fragments of source code and documentation (javadoc) in the graph nodes, as it can be seen in Figure 2.4.

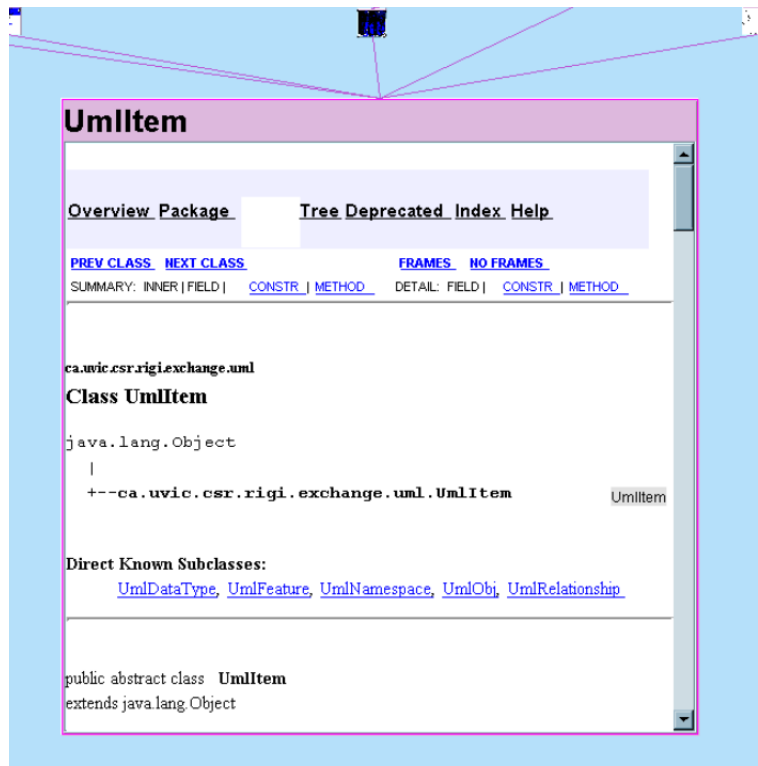


Figure 2.4: SHriMP view of a graph node, extracted from [65]

2.8 Summary

Program Comprehension (**PC**) is a Software Engineering discipline that studies the way programmers understand programs, though the analysis of the mental processes executed in order to do so. In particular, this area of **PC** is a major concern for the area of Software Maintenance, which by obvious reasons, demand the necessity of a faster program comprehension process. However, the difficulty on understanding completely this process arises from the fact that different programmers use different cognitive strategies according to different factors: the programmer itself, the program to comprehend and task of comprehension at hand. However, all of the different processes followed stumble into different problems which affect them.

In the process of comprehending a given program, the programmer maintains and updates a mental structure which represents the current state of knowledge about the program. This mental structure is defined as a *Mental Model*. To build such a mental structure, the programmer takes advantages of elements that facilitates that construction. These elements can be divided into static and dynamic elements and also cognitive enablers.

In order to comprehend a given program, a programmer also uses is own personal knowledge. This knowledge can be represented using a *knowledge domain*. In particular there are two types of knowledge domains which are of great interest for the **PC** research: the Problem and the Program Domain. The first represents the knowledge related with the problems the program at hand is concerned on solving. The second represents the knowledge related to the programming in general and with the programming language of the source code of the program in particular.

These different types of cognitive processes, are subject of intense discussion between several authors of the **PC** area. In particular, several authors have focused on finding models that programmers follow in order to understand program, and defined them as *Cognitive Models*. Each model has its own characteristics, however they can be divided into three different groups: top-down, bottom-up and hybrid models.

Considering the importance of the process of comprehending programs, several authors have taken advantage of the research in **PC**, and adapt it to the development of tools and approaches that help programmers understand faster a given program.

Chapter 3

Information Retrieval for Program Comprehension

In the previous section it is described the fundamental cognitive models that several **PC** researchers have theorized to explain the programmer's cognitive processes behind the comprehension of a program. Although the essence of these models is still very present in actual **PC**, the reality is that the most recent one is dated from 1995, raising the question if whether these models have accompanied the evolution of software.

Since that time, software has become larger and more complex. As *Software Maintenance* has become more demanding, the process of comprehending a program should follow this demand, reducing the time and effort to complete it. To accelerate this process, programmers choose to follow an *as-needed strategy* (Chapter 2) instead of trying to understand the whole program. In [78], the authors corroborated this theory, and completed it. According to *Rajlich and Wilde*, when the program is too big to try to understand it completely, programmers prefer to understand minimal parts of the program, parts which are defined by *concepts*. They also gave a definition of *concept*, presenting it as *units of human knowledge that can be processed by the human mind (short-term memory) in one instance.* [78].

However, the idea of finding concepts on code is not new. *Biggerstaff et al.* identified it and called it the *concept assignment problem* [14]. The authors defined this problem as the *problem of discovering (...) human oriented concepts and assigning them to their implementation instances within a program* [14]. Several researchers have addressed this problem, by creating approaches and tools to assist *concept location*. Although there are enumerous types of approaches for *concept*

location, this problem is mostly addressed in the literature using techniques from *IR*.

In this chapter there is addressed the subject of the role of Information Retrieval techniques on **PC**. This chapter starts with Section 3.1 that gives an overview of the **IR** area, introducing fundamental definitions, detailing the main process behing any **IR** system, and includes the enumeration and full description of the main **IR** models focused on the literature. This chapter ends with Section 3.2, that includes an exhaustive study about the role of **IR** on **PC**, introducing and describing several approaches that address this problem.

3.1 Information Retrieval

IR is defined in [61] as a *field of study* that has the goal of *finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)*. Despite of this definition, according to *Baeza-Yates and Ribeiro-Neto* [8], **IR** has expanded its research, from merely indexing and searching of documents, to modeling, document classification and categorization, systems architectures, amongst other subjects.

Although the area of **IR** has born many years ago, the sentiment was that **IR** techniques were exclusive of some restrictive groups, which included librarians and information experts. The use of **IR** techniques have become very popular in the past two decades, due to the creation of the *World Wide Web (WWW)*. The large propagation of information in the *WWW*, demanded the development of tools that responded to the users requirement for the search of specific information. This led to the creation of *search engines*, as the world renowned *Google*[1].

As was mentioned before, **IR** is mainly a task of retrieving documents responded to a given query. There are many models [28, 40] defined in the **IR** literature that address this task, but the base process in **IR** is well defined. But before addressing the process behind **IR**, it it essential to discuss the factors that affect the success of an **IR** system, which are: the *user's task* [10] and the *logical view of the documents* [8].

In an **IR** system, the task of the user is to translate its necessity for a specific source of information or document, in the form of a query that the **IR** system can interpret (typically, a set of keywords). According to the query, the **IR** system *retrieves* a set of documents, which it considers

[1] http://www.google.com

to satisfy the conditions, and *ranks* them according to its interpretation of relevance to the query.

Typically and due to performance reasons, **IR** systems see documents as a set of index keywords (or *inverted index* [61]) that can be manually or automatically attributed. In the database of an **IR** system, the information of the documents is stored in the form of a *dictionary* [75]. This structure contains the set of all words that appear in all the documents, and to each word is associated the list of documents in which the word occurs and the respective weight in the document (typically the frequency). This represents the *logical view of the documents*.

In this section is described the base process of **IR**. This process is the matrix for the variety of **IR** models available in the literature, that are also addressed in this section.

3.1.1 The Information Retrieval Process

The base process or general model in **IR** is shown on Figure 3.1, and is composed by two stages: *indexing* and *retrieval*.

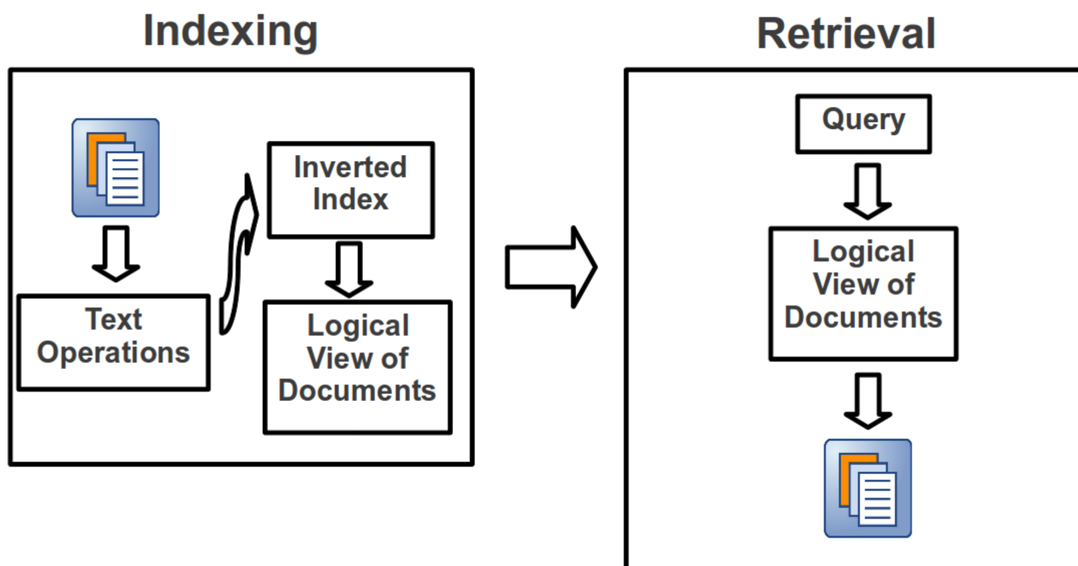


Figure 3.1: Information Retrieval system main process

On the first stage, the *logical view of documents* must be defined. The collection of documents that form the database of the **IR** system must be properly indexed (using the *inverted index*), for a good search and retrieval performance. To create the index, each document must be processed to extract the keywords that form it. Typically, not all of the words included in the document take

part of the index. There is a filtering process, using *text operations*, that pretend to extract the most concise set of keywords that form the index. These text operations include:

Sentence tokenization Breaking the text into individual sentences.

Word tokenization Breaking the sentences into individual words.

Stemming Reducing a word to its grammatical root.

Stop word removal Elimination of words that are too frequent or do not have any significance or value.

After the application of *text operations* to extract the indexes for the documents, it is then possible to create the *dictionary*, by calculating the weight of each word in the document. Apart from the typical and simple attribution to the weight of the frequency of the word in the document, there are several weightning algorithms. However, the most cited and used in the literature is the *Term Frequency - Inverse Document Frequency (TF-IDF)* [50, 80]:

Term Frequency - Inverse Document Frequency In this method, it is assigned a higher weight to words that occur frequently on a small set of documents. On the Equation (3.1), there is presented the **TF-IDF** formula to calculate the weight ($w_{i,j}$) of the word j in the document i, and where $tf_{i,j}$ is the frequency of j in i, N is the total number of documents and df_j is the total occurence of j in all documents.

$$w_{i,j} = tf_{i,j} \times \log N/df_j \tag{3.1}$$

The second stage, *retrieval*, starts with the definition of a query by the **IR** system user. Then, the same *text operations*, mentioned before, are also performed on the query, in order to be in tune with the *inverted indexes*. After the query is preprocessed, the search phase begins. During this phase, all the documents that contain at least one word from the processed query are retrieved. The last phase of this stage, is the ranking process, which evaluates each document according to the relevance to the query. The documents are then presented to the user, in a descendant order, according to the attributed rank value.

In the **IR** literature there is a vast number of **IR** models that instantiate this general model, specially by altering their ranking algorithms. In the following sections, the most relevant ones are briefly described.

3.1.2 Vector Space Model

In the *Vector Space Model (VSM)* [83, 31, 50] documents and queries are seen as vectors of weights of words (*bag-of-words*), as it is shown on Equation (3.2). d_i represents the vector for document i and $w_{i,n}$ represents the weight of the word n in i.

$$\overrightarrow{d_i} = (w_{i,1}, w_{i,2}...w_{i,n}) \tag{3.2}$$

VSM defines the rank of a document, by calculating the similarities between the document and the query vectors. This is done using the *cosine similarity*, that calculates the cosine of the angle between the two vectors. The cosine similarity formula is enounced on Equation (3.3), where d_i is the document i, q is the query, and Θ is the angle between the two vectors.

$$sim(d_i, q) = \cos\Theta = \frac{\overrightarrow{d_i} \cdot \overrightarrow{q}}{|\overrightarrow{d_i}||\overrightarrow{q}|} \tag{3.3}$$

If the document and the query do not share any words, the attributed rank will be of 0, according to the *cosine similarity*. This value will be higher, as the angle between the vectors decreases.

3.1.3 Latent Semantic Indexing

Latent Semantic Indexing (LSI) [28, 44] is the application of *Latent Semantic Analysis (LSA)* [47, 30, 48] to **IR**.

In **VSM**, the meaning and semantic of the words and the relationships between them are not taken into account. They are only simple lexical objects for matching. This, however, has created two major issues that cause lack of effiency in **VSM**: *synonymy* (different words with equal meaning) and *polysemy* (word with multiple meanings). While the first problem drives to a poor *recall* (fraction of the relevant documents that are retrieved), the second one causes a lack of *precision* (percentage

of retrieved documents that are relevant to the query) in the retrieved documents.

To solve these problems, **LSA** predicts that there is a *latent* but *hidden* structure behind the relationship between words. This latent structure preconizes the existance of latent variables, which are seen as *concepts*. The goal of **LSA** is to associate these hidden variables (*concepts*) to the observed ones (words and documents). To achieve this, **LSA** builds a new dimension space, the *concept space*, where documents and documents can be mapped. Words and documents who are close to each other in the *concept space*, are seen as related, and can represent a given concept.

To build the *concept space*, **LSA** uses *Single Value Decomposition (SVD)* [3] which is an algebraic technique that factorizes a matrix in order to enable the reduction of its dimensions, without important losses to the original matrix. In this context, the **SVD** algorithm will not be described due to its extension and demand for advanced algebraic and mathematical background. However, the full description of the algorithm can be seen in [3]. For now, it is important to focus on the result of the **SVD**.

The application of **SVD** on a matrix, factorizes it into three matrixes, as shown on Equation (3.4). A is the original matrix (t lines $\times d$ columns) and r is the number of chosen dimensions.

$$A_{t,d} = U_{t,r} S_{r,r} V_{r,d}^T \tag{3.4}$$

In the context of **LSA**, the first step is to build a matrix that contains the information from the documents. Each document vector represents a column in the matrix, and each line represents a word. As was said before, the application of **SVD** to the matrix, will factorize it into three. Then it has to be chosen a suitable number of dimensions for the new *concept space*, to reduce the matrixes. Considering again the Equation (3.4), each line (vector) of the matrix U will give the position of each word in the *concept space*. The same principle is applied to the matrix V, but in this case the position of each document in the *concept space* is given by the respective column (vector).

To perform the retrieval, the query must also be represented in this *concept space*. In this case the value of each word for the new query vector in the *concept space*, is equal to the total sum of all the values (the columns) correspondent to that word in matrix U.

As the query, words and documents are represented in the same *concept space*, it is possible to calculate similarities between these entities, in particular ranking the documents in relation to the query. This can be done, amongst other methods, using the *cosine similarity* (Equation (3.3)).

3.1.4 Probabilistic Latent Semantic Indexing

LSA uses **SVD** to factorize the word by document matrix. However, **SVD** has a high computational complexity ($O(min(t^2d, td^2))$), which causes poor performance to **LSA**. This is aggravated by the fact, that the **SVD** must be recalculated at every change on the matrix (e.g. new documents). Furthermore, **LSA** has only a purely mathematical foundation.

To overcome these problems, and to include a statistical foundation on **LSA**, *Hofmann* developed the *Probabilistic Latent Semantic Indexing (PLSI)* [40], that does not use **SVD** to approximate the word by document matrix to model relationships between words and documents, but uses a statistical model called *aspect model*. This model is a *latent variable model, for co-occurrence data which associates an unobserved variable with each observation* [40]. This unobserved variable is defined as a *topic* or *concept*, as mentioned before.

The goal of **PLSI** is to calculate $P(w|t)$ and $P(t|d)$ knowing only $P(w, d)$, which are respectively the probability of the occurence of a word w on a given topic t, the probability of the existence of a topic t in a document d and the occurence (weight) of the word w in the document d. The mathematical model of **PLSI** is enounced on Equation (3.5).

$$P(w, d) = \sum_{t \in T} P(t)P(t|d)P(w|t) \tag{3.5}$$

To solve this problem, **PLSI** can use several methods, such as the *Maximum Likelihood* or the *Expectation-maximization* algorithm, which are methods for estimating the parameters of a given *latent variable model*. Like **SVD**, these are complex algorithms that demand advanced mathematical background, so they are not described in this document. However, the details of the algorithms can be seen in [40, 67, 29].

To perform the retrieval of documents, the query is seen as a new document to include in the matrix. This will need, of course, the recalculation of the probabilities, using the estimating algorithms. Similarity measures, like the *cosine similarity*, can then be used to perform the ranking of the documents according to the query.

3.1.5 Latent Dirichlet Allocation

In **PLSI** the distribution of topics on documents is purely uniform. However, thinking about the reality, this is almost always not true. Documents from the document database of an **IR** system can contain different numbers of topics or concepts.

To translate a better view of the reality on the distribution of topics or concepts on documents, *Latent Dirichlet Allocation (LDA)* [15] defines the distribution of topics as following a *Dirichlet distribution* [57]. The mathematical model behind LDA can be seen on Equation (3.6).

$$P(w,d) = \sum_{t \in T} P(t|d)P(w|t) \tag{3.6}$$

$P(t|d)$ and $P(w|t)$ follow two *Dirichlet distributions*, respectively θ, a distribution of topics over a document, and ϕ, a distribution of words over a topic. The estimation of these distribution is seen as a problem of *Bayesian inference* [16]. There are several algorithms to solve this problem, including the *Gibbs sampling*, that is fully detailed in [90].

The formula to calculate the ranking of a document d according to a given query Q in **LDA**, is enounced on Equation (3.7) [56].

$$sim(Q,d) = P(Q,d) = \prod_{q_k \in Q} P(q_k, d) \tag{3.7}$$

3.2 Information Retrieval For Program Comprehension

In the last decade, the use of **IR** techniques on the development of **PC** tools has risen substancially. Typically, all of the research done so far in this field has the goal of addressing the *concept assignment problem*, which is, as was mentioned before, the verification that real-word concepts (Problem Domain) are integrated on the source code (Program Domain), and therefore, there is the need to find those mappings. In this section it is described the state-of-the-art on the role of Information Retrieval on **PC**.

The tools and approaches, described in this section, follow the same general process, similar to the described in Subsection 3.1.1, and mentioned by *Marcus et al.* in [64]. This process has the

following steps:

1. Preprocessing of the source code or the documentation to create the document database (e.g *text operations*);
2. Indexing the created documents;
3. Formulation of queries interpreted by the **IR** system;
4. Preprocessing of the formulated queries;
5. Execution of the queries;
6. Retrieving and ranking of the results.

This type of **IR** process was introduced in the **PC** field by *Maletic and Valluri* in [60]. In this work, the authors applied **LSA** to create clusters of similar source code components. The performed text operations on source code were rather limited, and included only the removal of non-essential symbols such as comment delimeters, syntactical tokens and ubiquitous tokens such as semi-colons. The granularity issue was addressed, as the authors decided to define mostly source code files as documents, because they felt that modules were too large in granularity and functions were too small. The new *concept space* in **LSA** included 250 dimensions, which was a large reduction to the original (approximately 2000). To create the clusters, the authors defined a similarity value λ, calculated by the *cosine similarity*. According to them, a document belongs to a given cluster if it is at least λ similar to any one of the other documents in the cluster. Although the authors assumed that the use of **LSA** does not take advantage of word ordering, syntactic relations or morphology, the results showed that **LSA**, applied to the source code, was able to induce clusters of source code components which were similar in reality. This work included two experiments which applied **LSA** to a C++ application and a C application. The clusters created for the first one (object oriented) reflected groups of related classes which addressed the same concepts or solved similar types of problems. For the second one, the created clusters seem to reflect classes, as most of them included one or two definitions of data structures and included functions that operated on those structures.

The previous work was continued in [58], as the authors studied the way the clusters were or not able to help on the process of comprehending a program. This study focused on the definition of

metrics that try to measure the similarity and cohesion between files and clusters. In an experiment, these metrics were applied on an application, and showed that these can identify groups of files that implement:

- Abstract Data Types or classes that represent concepts;
- a general or abstract concept;
- a general structure (lists, arrays,etc.) or similar ones.

The next step followed by the authors was to combine this method with structural ones, in a work described in [59], to show that it can help on program comprehension in two different dimensions. The structural methods included the data-flow, control-flow analysis, coupling, etc. Again, metrics to assess the cohesion between files and clusters were identified, this time including the structural information too. In the experiments, the metrics returned similar results from those from the previous work.

In [62], the same authors continued their work of applying **IR** techniques for **PC** purposes, by defining an approach that retrieved links between documentation and source code using **LSA**, and compared it to another approach that used VSM [6], from *Antoniol et al.*. In this work, source code and documentation were represented in the same conditions. As usual, the granularity chosen for source code was at file level; for documentation the chosen granularity was at section level. The performed text operations were a little improved, and included the splitting of identifiers, that follow usual naming standards, into separate words (e.g. "blueCar" or "blue_car" divided into "blue" and "car"). To retrieve the links between documentation and source code, the cosine similarity between each pair from the document matrix and source code matrix were calculated. The authors defined a threshold of ϵ, which represented the minimum value of similarity to establish a link between the source code and a document. The results from the experiments performed on a real application showed that the traceability reached a 100% recall value, and approximately 12% for precision, which were superior values from those obtained by *Antoniol et al.*, using VSM.

The concept assignment problem was addressed by the same authors in [63]. In this work, the aim was to use the **LSA** model to map problem domain concepts on the source code. These concepts were found using queries expressed by the user or were created automatically. The process behind this work followed the same patterns as the previous ones. However, the granularity of

documents was rather different. In the experiment they conducted on a C application, each declaration block, function and header file was considered as a document in the document database. To search for a particular concept, the authors defined several manual queries, ones more complete than the others. This approach also included the existance of automatic queries, which were developed using a single word or phrase indicated by the user, and the 40 terms more related to that word or phrase, were automatically added to the query by the system. The first type of queries, the manual ones, resulted on a 100% recall and a 33.33% precision. The automatic queries retrieved also 100% recall and a 26.66% precision.

The works previously described have mainly focused on experimenting concept location on C programs. In [64], the authors studied the way concept location affected the comprehension of object-oriented programs, which by its nature, are already design into classes which represent, more or less, concepts from the Problem Domain. This study addressed the concept assignment problem using several static techniques of code analysis: *grep* (string pattern matching), dependency search and **LSA**. In the experiments, the results showed that even on object-oriented programs there is the need to take advantage of concept location techniques, due to the fact that is not straightforward that a class hierarchy represents a hierarchy of Problem Domain concepts, as only a minority of these concepts are implemented as specific classes, and the rest are scattered over several classes. The authors also concluded that the simple fact that a program is object-oriented does not simplify the task of concept location.

On the object-oriented programming paradigm, modularity is considered to be the most important pattern. However, by performing modularity there are some functions or classes, like persistence and logging, that cut across or serve across different other classes, as concluded in the previous described work. These are known as *crosscutting concerns*, and on **PC** context these types of concerns, implementations of a concept, pose difficulties on the understanding, not only because the implementation of a given concern is not reserved to a single module but also because it becomes more difficult to understand the single concern of a single module. To solve this, *Aspect mining* can do two things: semi-automatically help developers understand an isolated concern, by executing a bottom-up understanding; or refactor an object-oriented programming system to another system that captures crosscutting concerns, which is called an *aspect-oriented system*. There are several techniques for *Aspect mining*, such as the fan-in analysis or dynamic analysis, but in this context of **IR** for **PC** the focus is on the *Formal Concept Analysis (FCA)* and other **IR** techniques.

FCA is a mathematical lattice theory that defines concepts as maximal groups of elements that share the same properties. Concepts are composed by its intension and extension. The intension of a concept contains the properties shared by the all the elements that composed it also known as the concept extension. With this information, it is possible to build a lattice where in the root or bottom concept are colocated the elements that share all the properties and in the top concept are colocated properties shared by all the elements.

In [76], the authors tried to address the concept assignment problem by combining **IR** techniques with FCA. The main idea was that the user searches for concepts, using **LSA**, which retrieve a list of the most relevant source code elements, and then it is applied FCA to create a lattice of concepts. The granularity chosen in this context was at method level. To apply FCA, the results from the query created by the user, are analyzed, and the top k attributes derived from them are selected. FCA then, does its part. The results, from the experiments, showed that for small programs with a small number of methods the vizualization of concept lattices is quite effective in **PC**, by showing the relationship between concepts or topics.

3.3 Summary

IR is a discipline of Software Engineering that emerged on the creation of the *World Wide Web*. It is focused on the search and retrieval of unstructured documents, responded to a necessity, expressed in the form of queries.

On **IR** there are two factors that affect its sucess: the user's task which is the translation of the necessity of the user into the form of a query that the **IR** system can interpret; and the logical view of the documents, which is the defined structure that represents the database of the documents in the system.

The process behind a typical **IR** system is divided into two stages. The first stage is called *indexing* and corresponds to the definition of the logical view of the documents. The second stage is called *retrieval*, which is defined as the stage that starts with the creation of a query and its processing, and ends with the retrieval of documents that responds to that query, properly ranked.

Several models have instantiated this one, however, in the literature, the most referenced ones are the **VSM**, the **LSA**, the **PLSI** and the **LDA**.

One of the main concerns of this **IR** area is to find solutions to enhance **PC**. In particular, the

developed approaches focused on addressing the *concept assignment problem*, which is the search of problem domain concepts on the source code.

Part III

Comment Analysis for Program Comprehension

Chapter 4

Source Code Comments

The previous part has the goal of giving a full description of the **PC** field, and includes an overview of the use of **IR** techniques to help on the understanding of a program. Typically every approach found on the **IR** for **PC** state-of-the-art builds the document database using terms from comments and identifiers. This is explained by the fact that these two different entities provide the semantic side of source code, through the content of natural language terms which are familar to the programmer and are contextualized in the Problem Domain. It it through them that the search for concepts on code can be effective and successfull.

Putting aside semi or fully automatic techniques for **PC**, *source code comments* are probably the best source of semantic information that can provide straighforward and quick answers about the Problem Domain concepts behind a piece of code, without the use of any tool. Moreover, this is the main goal of a comment. Comments have the *sole purpose of aiding the human reader* [97]. As mentioned before, comments contain terms of the Problem Domain written in a natural language. Furthermore, comments can be composed by sentences which structure these terms in an effective way to aid on the comprehension of the respective piece of code.

However, there is a lot of controversy about the content of comments and also about the use of comments on source code. Comments can contain the explanation of the source code and also the way to use that source code (in the case of a function/method or class comment). However, there are those, such as the apologists of Extreme Programming (XP), which preconize the creation of self explanatory source code, instead of using comments.

The controversy of using or not using comments is not relevant in this context. The important at the moment is to see comments as a natural tool that help understand programs.

In the literature, the study and analysis of comments has not been very extent to the moment. However, in this chapter, it is given an overview of the characteristics of the comment as an element of source code and as a documental piece, as available in the literature. This chapter starts with the enumeration of the different types of comments on Section 4.1. Then on Section 4.2 it is presented a taxonomy of comments on terms of different types of content. On Section 4.3 there is presented a small study of the characteristics of the language normally used to write the contents of comments. There is also included an overview of the literature of the relations between comments and the size of the program, on Section 4.4. This chapter ends with Section 4.5, that includes a small study about the available literature on the subject of comment density and the practice of commenting on the development of software.

4.1 Types

A comment is one of the elements of the document structure of the source code [97], which is defined as the set of elements which do not take part of the programming language in question. These also includes white spaces (indentation, etc.) and choice of names or identifiers. As was mentioned before over comments, this structure has only the main goal to provide information that can be relevant for program understandibility.

In terms of types of comments, they can be divided into two types: *line* or *inline comments* and *block comments*. The only differences between these two types is the number of lines they can contain and the syntax to define them. The inline comments can only contain one single line, whereas block comments can contain one or more lines. Each programming language can provide one or both types of comments in its language grammar. The *Java*™programming language [38], for example, provide both types of comments in its syntax, and also a variation of the block comment called *Javadoc* [46]. This special type of comment, can only be used to comment classes and methods, and is defined by a special syntax, which includes a set of tags, such as the author, the version, the parameters of the method if applicable, amongst others. This type of comment can be then preprocessed by the Javadoc tool, which builds an HTML file with the API of the source code in question.

Each programming language can vary on the syntax given to the definition of each type of comments. Considering for example, the Java programming language, it defines inline comments

as shown on Listing 4.1.

Listing 4.1: Inline Comment

```
// Text of the Comment
```

As for the block comment, Java defines them as shown on Listing 4.2.

Listing 4.2: Block Comment

```
/*
 Text of
 the Comment
*/
```

An example of a Javadoc comment can also be seen on Listing 4.3.

Listing 4.3: JavaDoc Comment

```
/**
 * Comment of method example
 *
 * @param param1 Parameter of the method
 * @param param2 Other parameter of the method
 * @return void
 * @author Author of the code
*/
void example( int param1, int param2 )
{
 ...
}
```

4.2 Content

As was mentioned before, there is still a lot of controversy around the content of comments, which raises the question of what is a bad or good comment. Apart from the controversy, a *bad comment* can start from being a comment which is inconsistent with the code which is commenting, and that leads to the misleading of the person who reads it [94].

In [71] the authors explored the content of comments on operating system code, and defined a taxonomy for comments. Below there is presented the most relevant cattegories and subcattegories of this taxonomy:

Type Includes the following subcattegories: *Unit* and *IntRange*.

The *Unit* subcattegory includes comments which indicate the unit of a variable. For example, a variable which represents a value of capacity, can be commented indicating that the capacity is defined in litres or centilitres (Listing 4.4).

Listing 4.4: Unit Comment

```
1 float capacity; /* capacity in litres */
```

The *IntRange* subcattegory includes comments which indicate the range of a variable. For example, a variable which represents a value of an angle in degrees, can be commented indicating that the range for the variable is from 0 to 360 degrees (Listing 4.5).

Listing 4.5: Range Comment

```
1 float angle; /* angle from 0 to 360 degrees */
```

Interface Includes the following subcattegory: *ErrorReturn*.

The *ErrorReturn* subcattegory includes comments which describe the return values from a function that indicates errors.

Code Relationship Includes the following subcattegories: *DataFlow* and *ControlFlow*.

The *DataFlow* subcattegory includes comments which indicate the dependance of data from variables or functions.

The *ControlFlow* subcattegory includes comments which may indicate:

- a piece of code not reachable;
- a missing break statement is intended;
- which function should be the caller of another function;
- the order of invokation of a given group of functions.

PastFuture Includes the following subcattegory: *TODO or FIXME*.

The *TODO or FIXME* subcattegory includes comments which indicate tasks that have still to be completed or enhance code which has errors.

Meta Comments which include information regarding copyright notices, authors, dates, amongst other types of meta information.

Explanation Includes all the types of comments which can not be classified by the previous cattegories and subcattegories.

The first four cattegories of comments were defined by the authors as *exploitable comments*, due to the fact that they can be used by other tools, such as bug detection tools and editors.

4.3 Comments and Language

It was already discuss that comments contain natural language terms structured in sentences of that same natural language. In [34], the authors examined the language included on comments. As a matter of fact, comments who are almost written in the English language, include only a part of that language, called a *sublanguage*. For the record, a *sublanguage* contains a limited set of the vocabulary from the main language, restrictions on syntactic and semantic constructions, as they tend to be repetitive. According to the study, the sublanguage of comments has the following characteristics:

- the present tense, indicative or imperative mood, is almost always used;
- limited set of verbs which include: is, uses, provides, implements, reads, writes, etc. In resume, verbs mainly from the program domain.
- personal pronouns are rarely used;

The terms included on comments was also studied in detail. In [39] the authors explored the existance of Problem Domain terms on the comments and identifiers of several programs. This study was basically developed by creating a list of Problem Domain terms, the authors though were related to the programs in study, and measuring the percentage of terms included on comments

and identifiers. The results showed that about 23% of the Problem Domain terms of the lists (135 terms) appeared on the comments alone, whereas only 11% appeared on identifiers.

4.4 Comments and Program Size

The importance of comments for understandibility purposes is inquestionable. However, on large production of software, where the understandibility is a key factor, it may not be easy to continue commenting and maintain the comments updated, throughout the evolution of the source code. To access this matter, the authors in [36] analyzed the co-evolution of comments and source code on several software systems. The main conclusions of this study were the following:

- comments and source code rarely co-evolve in time, as new added source code is barely commented;
- when there is a co-evolution of source code and comments, mostly classes and methods are targeted for commenting;
- in 97% of the time, comment changes are triggered by source code changes.

However, the conclusions of this study does not coincide with the conclusions of a similar study, described in [41], which accessed the evolution of comments on *PostgreSQL* [66] source code, particularly on function declarations. Contrary, to the previously described work, this study concluded that the percentage of commented functions maintained consistent throughout the evolution of the system.

4.5 Comment Density and Practice

As was mentioned before, there is a lot of controversy around the use and amount of comments on source code. On those who support the use of comments, there is not a clear idea of how many comments should a program contain. As there is no standards on the use and amount of comments, software communities have followed their own principles regarding the commenting practice and density. In [7], the authors analyzed the practice of commenting on a particular software community, open source, in order to access the principles of commenting followed by this

community. The first conclusion of this study is that about one line of code in five lines (around 19%) is a comment line. The second conclusion, which was related to the influence of programming languages on the commenting practice, showed that *Java* programs tend to be more commented that programs from other programming languages. The other conclusions, show that the comment density are independent from the development team size and from the project size.

Another similar study, described in [4], analyzed the relation between the comment density in a project and the stability of this one, throughout an empirical experiment. The results from this study showed that projects with a higher level of comment density have more tendency to be more stable throughout the updgrades.

In the same way, the authors in [91] performed a study about the quality characteristics of open source development. To do so, they used a proprietary set of tools, called Logiscope©, that apart from calculating and measuring these characteristics, also contains its own programming standard that is the result of the analysis of millions of source code lines of industrial applications, and it is known that several large corporations use these standard to control their software development. One of these standards is the comment frequency or density, that quantifies a factor of quality called *self descriptiveness*, which is one of the principles of *ISO 9241*. The rule, for Logiscope©and for this study, was that this frequency must be at least of 20%. This value is much alike with the results from concluded in [7], which is of one line of comment per five lines of code.

4.6 Summary

A source code comment is an element of source code, whose function is considered to be of only providing information for understandibility purposes. However, on several sectors of the Software Engineerin there is still controversy and discussion about the real utility of the comments.

On terms of syntax, apart from the fact that are different types of comments according to the programming language at hands, they can be divided into two different groups: the inline comments and the block comments. The first only permits the introduction of one line of comment, whereas the second permits several lines.

On terms of content, there is also a taxonomy for comments. In general, comments can divided into six different groups on terms of content: *type*, *interface*, *code relationship*, *past future*, *meta* and *explanation*.

There was also identified that comments, in general, have a particular language that is considered a *sublanguage* of a certain natural language, such as the English one. In particular, this language have a limited set of verbs and tenses that are used, and personal pronouns are almost not used.

Although commenting is a good practice it may not be so easy to do on large production of software. A study have shown that in general there is not a pair evolution between source code and comments. However, there was also an other study that contradicted this previous one, by presenting proves of this co-evolution.

There is not a standard for the amount and percentage of comments on the source code. A study in this question, have defined that on open-source project, that tend to include high percentage of comment, and a high percentage of comment quality, almost 19% of the source code was for commenting. There was also a study that concluded that programs with a high density of comments tend to be more stable throughout time.

Chapter 5

Comment Analysis For Program Comprehension

The previous chapter gives an overview of the characteristics of comments as source code elements, as it is seen on the literature. As was mentioned, comments have the goal of helping the programmer understand unknown or unfamiliar pieces of code, and in general, to help understand the hole program. Therefore, comments may be considered a native and natural **PC** tool.

However, saying that the goal of comments is to understand programs is not sufficient to garantee that this is attained. As was mentioned before, the quality of comments is directly related with the easiness on the comprehension task at course. So, for understandibility purposes it is necessary to garantee good comments.

The nature and content of comments is already discussed on the previous chapter. In this one, it is discussed if comments do help on comprehension and why. This subject is focused on Section 5.1, that includes several studies that explored that suspicion. Taking this, several works on **PC**, have taken advantage of the comments nature to produce **PC** approaches, apart from those described on the previous part, which use **IR** techniques. These works are enumerated and described on the end of this chapter, on Section 5.2.

5.1 The role of comments on Program Comprehension

Comments have indeed the incredible power to aid programmers understand code with or without read it. Considering this, several authors have conducted a number of experiments to prove the

importance and role of comments on the comprehension of programs.

One of the first experiments was conducted in 1981 by *Woodfield et al.* in a work described in [104]. This intended to measure the effect of comments and also of modularization on the understanding of a program. The principles of this experiment where the following: forty eight experienced programmers, graduated and undergraduate, were chosen to participate in this study; the programmers were divided into groups of eight; to each group it was given a different version of a Fortran program and an equal test quiz about the program; half of the groups contained versions of the program with comments and the other half contained versions without comments; to maximize the need for a comment comprehension approach every version contained meaningless identifiers and the indentation was removed. The results from this experiment showed that the groups which were given commented versions were more able to answer to a bigger quantity of questions correctly. The authors also pointed out that these results, regarding comments, were independent from the modularization, which indicates that the comments alone provided a significant help for program comprehension.

The experiments regarding the role of comments on **PC** continued in 1985 in [95]. In this work, the authors described an experiment which aimed at study the effect of comments and modularization in the *readability* of the Banker's Algorithm. In this case, the authors defined *readability* as the quickness and accuracy of critical information that a programmer is able to extract from a program. This experiment followed the same patterns of the previously described one. The results showed that only comments produced more *readability* for the experiment subjects regarding the algorithm at study. Another experiment from the same authors was conducted and described in [96]. Again, the results showed the *readability* gain that comments produce.

The most recent experiment available on the literature, regarding the role of comments on program comprehension is described in [70]. In this case this experiment contained slighter different goals from the previous ones. This experiment aimed at studying the differences between the understanding of programs using class or method comments. Again, the same patterns were followed to proceed with this experiment: there were versions of the same program, with both types of comments, with only one type of comments and without any comments; the subjects had to answer to a test quiz regarding the program at study. Corroborating the results from the previous experiments, the authors concluded that the versions with comments were better understood than those which did not contain comments. Regarding the differences between class and method comment under-

standing, the results showed that the subjects which had a method commented version performed better on the quiz than those which possessed a class commented one.

Apart from the experiments, several authors have addressed the subject of the role of comments on **PC**. Commenting on the theory that *the best documentation for a computer program (...)* includes *a smattering of enlightening comments* proffered by *Kernighan and Plauger* in [43], *Brooks* have also defended in [17] the importance of comments on the understanding of programs. In this case, *Brooks* goes further by defining comments as a mean of establishing a bridge between different knowledge domains, specially between the Program and Problem domain, mentioned and overviewed on the previous part. According to the author, the programmer who writes the source code must be aware of this fact, and when commenting, he should incorporate information which ease the establishment of this bridge between the two domains.

5.2 Comment Analysis Program Comprehension tools

In the previous part, it is described some approaches for tools for **PC** purposes which were based on **IR** techniques. In this case, these techniques took advantage of the content of source code to develop the document database, specially from comments and identifiers. However, apart from the **IR** approaches, there is a significant lack of tools that take advantage of comment information to automatize program comprehension.

The only approach available on the literature which is the exception is the one developed by *Vinz and Etzkorn* and described in [98]. In this work, the authors propose a comment to code matcher. One of the problems that the authors faced was the difference between the abstraction levels of both domains, and they had to do this mapping with some accuracy. They considered individual lines of code, as well as methods, classes or functions that had comments attached. To analyze comments, they took advantage of *PATRicia* [33], a system that applies natural language processing techniques to analyze comments and identifiers of a program. This system uses a sentence parser, that has information of standard comment syntactic structures and identifiers formats. It has also a semantic processor that uses a knowledge base which represents concepts in the form of a hierarchical semantical network. In order to do the code understanding, they took advantage of the same knowledge representation and inference engine of *PATRicia*. To do this mapping, they analyzed each domain individually. On the comment understanding, they detected occurrences of

words and infer relationships between them to detect concepts. On the code understanding, they analyzed the structure of code and the order and nature of the statements on program to infer some concept, or standard algorithm. In the end, if in both domains the same concept was found, they could consider that the comment matched the code. This solution lead to successful results, taking the authors to conclude that it could be applied to a *practical application size* [98].

5.3 Summary

The role of comments on **PC** is a major concern of several authors and researchers. In order to study the real power and role of comments as catalyzing tools of comprehension, these authors and researchers have executed several tests with human subjects. These studies aimed for the same goals and used the same principles: using versions of the same program, one without comments and another with comments, the subjects had to solve a quiz test about the program. The results showed that, in all the cases, the subjects with commented versions were more able to understand the programs fast and with more effectiveness.

More than simply stating that comments do help on comprehension, *Brooks* have enumerated the reasons comments can be effective for **PC**, by defining them as privileged sources of Problem and Program Domain information that ease the process of establishing bridges between the two domains.

Although comments have been confirmed as extremely useful for **PC**, lacks the variety of approaches that uses comment information to enhance **PC**, as they all address the problem using **IR** techniques. The only exception is an approach developed by *Vinz and Etzkorn*. This approach uses techniques such as the introduction of semantic processors, semantical networks and knowledge bases. Although the authors assume good results returned by this work, the last reference dates from 2008, and there is no sign in the literature that this project was updated by the original authors or by different authors.

Part IV

Darius

Chapter 6

Darius: The first stage

Throughout Chapter 2 it is analyzed the importance of **PC** on the maintenance of a given software system or program. This importance lead to the establishment of **PC** as a relevant area of study, in which several theories have emerged to try to explain the comprehension process. Taking in consideration some of those theories, several authors have taken advantage of them to develop approaches and tools that try to enhance **PC**. As it was seen, most of them are based on **IR** techniques (Chapter 3), that basically try to locate Problem Domain concepts on the source code, or was it was previously mentioned, the addressing of the *concept assignment problem*. This adaptation of **IR** techniques for concept location was mostly exclusive to the exploration of terms from comments and identifiers, which reflect the Problem Domain information needed to understand the semantic of a program.

In Chapter 5 it is addressed the subject of the role of comments on **PC**. The conclusion, from several studies, is that comments can provide help and facilitate the process of comprehending a given program. Taking this, the objective of the work described in this dissertation is to take advantage of the analysis of source code comments to provide information for the enhancing of **PC**. However, recalling Section 1.2, before developing such an approach, it is necessary to take in consideration two major issues: the quantity and the content of comments on source code. Without the fulfillment of these two factors, the development of a Comment Analysis **PC** tool is not feasible.

So, before proceeding for the development of the **PC** tool that analyses comments, it is necessary to study in detail these two factors by performing two different tests. In order to execute these preliminary tests that can prove the feasibility of the use of Comment Analysis for the development of a **PC** tool, the first step was to develop a preliminary version of that tool that only had the goal of

running those tests. To this tool it was given the name of **Darius**[1].

In this chapter the focus is on this preliminary study and on the results and conclusions extracted from it. It starts with Section 6.1 that identifies the goals and the strategies to follow in order to execute this study. On Section 6.2 there is introduced the first version of **Darius**, and the presentation and enumeration, with the proper description, of each one of the modules included in this version, as well of the graphical interface. This chapter ends with Section 6.3 that presents the steps that were taken in this preliminary study, and also includes the presentation and discussion of the results that were able to be extracted.

6.1 The Preliminary Study

As was mentioned before, the feasibility of the Comment Analysis **PC** tool was be dependent of two different factors: the quantity and content of comments. In order to explore them, it was executed two different tests that, respectively, tried to measure these two factors on a set of elements of study. In this section are enumerated the objectives of these two different tests and the process that has to be followed in order to properly execute the tests.

6.1.1 Comment Quantity

As it was referred on Section 4.5 there is no clear idea and no standards whatsoever about the right amount of comments on source code. However, considering the work [7] mentioned on that section, open source projects contain in average 19% of comments on their source code, and the programming standard of Logiscope©, also described in Section 4.5, also assumes a similar value of one line of comment per five lines of code. So, assuming that value as a reference, a program has a sufficient amount of comments, if they are at least 19% of the source code of the whole program.

Having established the objective of the test, it is important to define the strategy followed to execute it. In order to calculate the percentage of comments on a program, it is necessary to analyze each source code file of the program and count the number of lines and also the number of comment lines. Then it is necessary to sum all the global values of number of lines and comment lines for all the source files. So, the ratio of comments on source code will be given by the division

[1] Relative to King Darius I of Persia, the first known man to create the first bridge between Europe and Asia, on the Bosphorus strait.

of the total number of comment lines and the total number of source code lines.

6.1.2 Comment Content

Measuring the quality of the content of comments is not a straightforward task. In reality, that is not even totally possible considering that there is no clear idea of what is a comment with quality. However, recalling Chapter 4, a comment has only the objective of aiding on the comprehension of programs. So, if comments have only that objective, it can be generalized the notion that a comment's content with quality is one that provides effective and relevant information to the user or programmer, which can lead him to understand, better and faster, a given program. Although having a clearer notion of comment content quality, is it no yet possible to measure it with precision in the moment. To do so, it would be necessary to execute similar tests, as those described on Section 5.1, which had the objective of studying the role of comments on Program Comprehension using human test subjects. So, it is necessary to create a better instance of comment content quality that can be effectively measured.

It is important to pay attention to what is mentioned on Section 5.1, in which is stated that comments provide effective help on the comprehension process, as they are a privileged mean of establishing bridges between the Problem and the Program Domain of the program [43, 17]. Taking in consideration *Brooks*' idea, it can improved the previous notion of comment's content with quality, by defining it as type of content that enhances the comprehension of a program by providing information regarding the Problem and Program Domain and also the means to establish a bridge between them. With this clearer notion, it is now more practical to measure and analyze it.

So, in resume, the testing of the comment's content's quality on a program must focus on the analysis of the words on the comment, and check whether it is utilized information regarding the Problem and Program Domain, in the form of terms which can be contextualized in those domains. However, similarly to the comment quantity issue, there is no standardized value for the percentage of those type of terms that must be included on the comments.

Despite this problem, it can defined as a reference the results from a similar study [39], which is detailed on Section 4.3, that had the objective of calculating the percentage of Problem Domain terms on comments and identifiers on the source code. This study showed that comments from the studied programs included, in general, 23% of the Problem Domain terms of the created lists. Although this study had not studied the percentage of Program Domain terms, it can be assumed

that this frequency is at its minimum the same value.

And so, with this information, this preliminary study of the comment's content can be performed by calculating the percentage of Problem and Program Domain terms included on comments taking in consideration the values from the study mentioned before.

To perform this study, the strategy was be very similar to that described in [39]. For each program at study, it was created a list of Problem Domain terms by analyzing the Problem Domain in which the program is inserted. Then, a general list for Program Domain terms was also created taking in consideration the programming language used for each program. To analyze the percentage of terms from both domains that appear on the comments, each different word was saved in a structure, and in the end it will be counted the number of terms from the lists that are saved on the created structure.

6.2 Darius: Version One

The first version of **Darius** had only the objective of executing the preliminary test previously described. So, taking in consideration the requirements that these test demanded, this first version of **Darius** included the following components (Figure 6.1):

- a comment extractor that withdraws comments from source code files and a code associator, that associates each comment with the piece of code (classified according to the grammar type: class, method, statement, and so on) it is commenting;
- a statistics calculator that provides quantitative results regarding comments in a program;
- a comment words analyzer which computes the frequency of words on comments, information that can be used to identify the domain of each word;
- a graphical interface which can be used to visualize all the information provided by the other components.

Now, for the programming language that **Darius** is able able to analyze the comments from, if it is recalled Section 4.5, it is there mentioned a study that proved that in general Java programs are usually more commented that other type of comments. So taking in consideration this fact, the decision was that **Darius** would analyze Java source code files and its comments.

The following subsections describe in detail each of the components mentioned above.

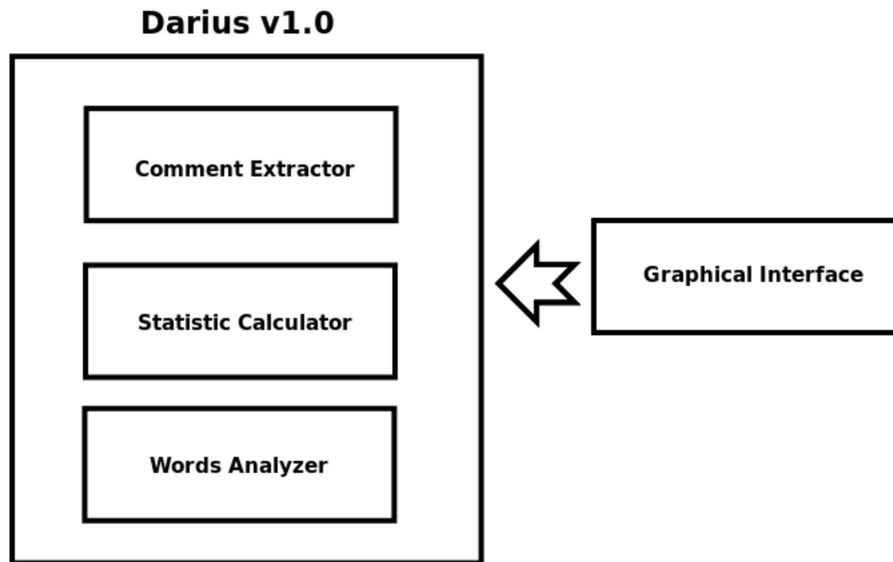


Figure 6.1: Darius First Version Structure

6.2.1 Extracting and Locating Comments

The first and essential component that **Darius** has is the comment extractor. In order to extract comments from a program or a software project, it is necessary to represent these entities first. To do so, **Darius** defines an abstract class called *Project* that can be instantiated on the class *JavaProject*. An object of this class represents a given program or software project. To build it, it is necessary to pass a path for the folder where the source code files of the desired program reside. **Darius** will then walk recursively through the folder defined in the passed path, looking for Java source code files, files with a ".java" extension. Each time **Darius** finds a Java source file, it creates an object of the class *JavaSourceCodeFile* that represents that given file, and adds it to a list associated with the *JavaProject* object. By doing this, in the end, this object will contain a list of all the Java source files that **Darius** was able to find. Whenever **Darius** builds a *JavaSourceCodeFile* object, the content of the file is extracted and associated to the new created object, along with the path to the file. The content of the file is analyzed in order to extract the respective comments. In the Java programming language there are three types of comments:

- InLine comments, **IC** for short, (//...)

- Block comments, **BC** for short, (/* ... */)
- JavaDoc comments, **JD** for short, (/** ... */)

These three types of comments are extracted from the source code using regular expressions. The use of regular expressions in spite the use of a parser, is explained by the fact that not all types of comments take part of the *Abstract Syntax Tree (AST)* (javadoc only) extracted by a parser. Although there are approaches [89] that try to associate inline and block comments with syntactic nodes in the AST, by changing the Java grammar rules, the extraction of comments in **Darius** pretends to be as generic as possible for every Java source code file. In order to discover and identify what type of source code entity is associated with the comment, the next line after the comment is extracted too. Considering the Java programming language, **Darius** associates comments with:

- *classes*,
- *interfaces*,
- *methods*,
- conditionals (*if*),
- loops (*while* and *for*)
- *switches*.

To execute this extraction, **Darius** contains a class called *JavaCommentExtractor* that extracts comments from a given input. This class contains a different type of regular expressions to every different type of Java comment. An object of this class receives an input text and searches for patterns from the various regular expressions. Whenever it is able to find one, it builds an object that represents the respective comment. This object is from the abstract class *JavaComment* and can be instantiated in one of the following classes, according to respective type of comment found: *JavaSingleComment*, *JavaBlockComment* and *JavaDocComment*. To build these types of objects, it is necessary to pass two strings: the content of the comment found and the next line after this comment. To every object of these three types of classes is associated an other object of the Enum type *JavaCodeAssociation*. This Enum type defines the code entity that is associated with the comment, and can be of the 6 types mentioned above. To identify these types, the *JavaComment*

class contains 6 different types of regular expressions that are used to search for the right source code entity in the next line string that was passed earlier.

So, in resume, in the end, **Darius** will have a *JavaProject* object that contains a list of *JavaSourceCodeFile* objects, each of whom will contain a list of *JavaComment* objects instantiated onto *JavaSingleComment*, *JavaBlockComment* or *JavaDocComment*. To each of this objects it will be associated an Enum type *JavaCodeAssociation*.

6.2.2 Comment Statistics

In order to develop the first preliminary study about the quantity comments on a program, **Darius** is able to calculate statistical values about the commenting practice on a program or software project. This preliminary study focused, more specifically, on the ratio between the number of comment lines and the number of source code lines. However, and taking in consideration that this would be a very simple task, that would present simplistic values, the decision was to enlarge the power of this statistic calculator by adding more statistical functions that could give important information about the commenting practice on a program. Thus, the **Darius** statistic calculator includes the following statistical functions:

- Number of comments of a project (global, per type of comment and per line of source code);
- Average number of comment lines per lines of code;
- Average number of lines of a non inline comment;
- Average number of each type of source code entity which is commented;
- Type of comments most used (global and per source code entity).

In order to calculate all of these statistics **Darius** has defined a class called *JavaProjectStatistics*. To build an object of this class it is necessary to pass a *JavaProject* object. This object will walk through all the *JavaSourceCodeFile* objects on the *JavaProject* object and the respective *JavaComment* objects, and will extract all the information it needs. In particular this *JavaProjectStatistics* will extract the following information:

1. total number of files;

2. total number of source code lines;
3. total number of comment lines;
4. total number of comments;
5. total number of inline comments, block comments and javadoc comments;
6. total number of each type of source code entity;
7. total number of each type of source code entity commented;
8. total number of each type of source code entity commented per type of comment;

With all of this information, this object can calculate the previously described statistical functions. For example, with the value of 4 **Darius** indicates the total number of comments on a given program or software project, and with the 5 this total number is specified by the different types of comments. By dividing the 3 value with the 2 one, **Darius** calculates the ratio of comment lines per source code line. These are just two examples of how **Darius** can calculate all of statistical functions previously mentioned.

6.2.3 Comment Words Analyzer

The second preliminary test focused on the analysis of the domain of the words included on comments, verifying if they are Problem or Program Domain oriented. The strategy created to perform this test is fully detailed on Subsection 6.1.2. In resume, this strategy involves the creation of 2 lists of words, each for a different domain, and also involves the accounting of those words that are present on the comments. Similarly to the idea presented on Section 6.2.2, this task was very simple, so the decision was to enforce the power of this analyzer by introducing other measures. So, generally speaking, the comment words analyzer can return the following values:

- Percentage and frequency of words in the list found in comments;
- Frequency of each type of comment that contains words from the list;
- Frequency of each type of source code entity commented that contains words from the list.

In order to perform all of these calculations, **Darius** has to know which words appear on the comments. To do so, there is a class called *JavaCommentsWords*, whose object must be build by passing a list of *JavaSourceCodeFile* objects from a *JavaProject* object. But before checking which words occur on comments, this *JavaCommentsWords* objects loads a list of "stop-words", words that are too frequent or that not have any significance. Then, this object walks through all the *JavaSourceCodeFile* objects, extracts the content of the respective comment and breaks it into a list of individual words. To each of the words from that list, it is checked whether the word corresponds to an identifier, particularly if it is written on camel-case. If this is the case, the word is broken into smaller words. Then, each word is passed to lower-case and is reduced to its respective stem using a proper English stemmer[2]. Then the resulting word is checked to see if it is a stop word or not. If not, the word is added to a structure that associates it to the current total frequency on the comment the word appeared, on the type of comment used and on the source entity commented. In the end, **Darius** will have the complete list of words contained on the comments of the program or software project.

To perform the analysis of the domains *per se*, **Darius** provides a class called *DomainWordsAnalyzer* that receives a *JavaCommentsWords* and a set of words. The *DomainWordsAnalyzer* object, first reduces every word from the set to its respective stem. Then, with the resultant new set of words, it walks through them and verifies if each one is included on the *JavaCommentsWords* object. If so, there is a variable that represents the total words of the set found that is incremented. Then, to perform the more advance calculations, it is also accounted the types of comments and the type of source code entities commented using that word, in all the comments associated with the word. In the end, this *DomainWordsAnalyzer* object will contain the number of words from the set that were present on the comments, the total occurrence of those words, and the total occurrence on the different types of comments and the different types of source code entities commented. With these values, **Darius** is able to present the functions mentioned above.

6.2.4 Graphical Interface

In order to execute these preliminary tests with more efficiency and to increase its pace, the decision was to create a graphical interface for **Darius**. This graphical interface pretended to be the most simplistic one, as it was not important to pay attention to extreme design details, that would just

[2]Snowball Stemmer by Martin Porter, http://snowball.tartarus.org/

spent valuable time with no special gains whatsoever. After some time of though on the design of the graphical interface for this first version of **Darius**, the decision was to focus on three important components that should be incorporated: the project or program loader, the comment statistic calculator and the comments words analyzer.

So, after developing the graphic design of these three components the result could be incorporated on the graphical interface of the first version of **Darius**. The final result can be seen on Figure 6.2. As it can be seen on this figure, the three components mentioned above, are the main focus of this graphical interface. On the top, there is the design of the project loader component as it can be seen on Figure 6.3. If the user clicks on the *Load Project/Program* button, a window opens as it can be seen on Figure 6.4. In this new window, the user can choose a folder where lies Java source code files, which of whom he wishes to be analyzed by **Darius**.

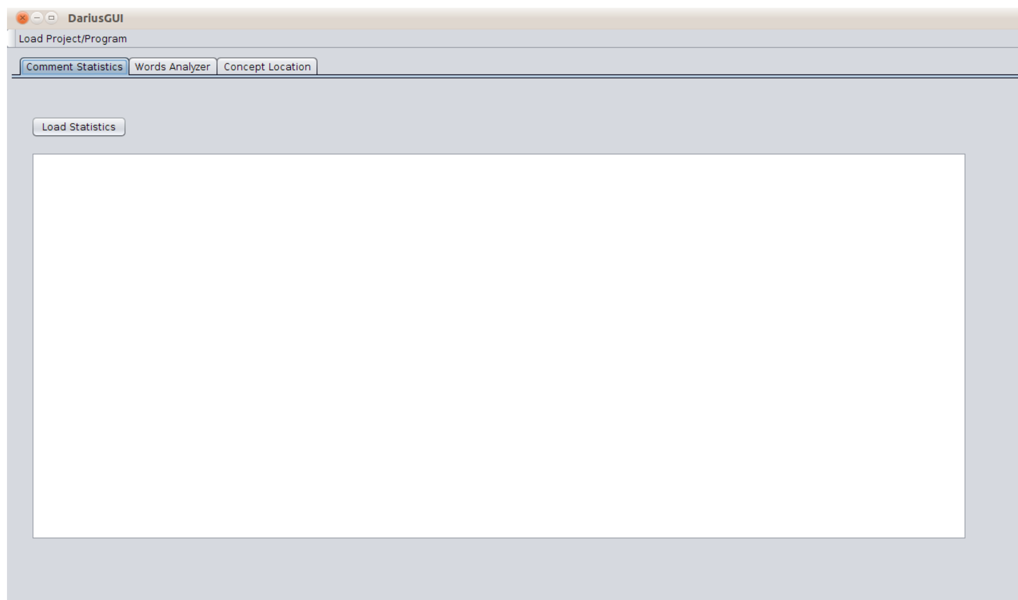


Figure 6.2: Darius Graphical Interface

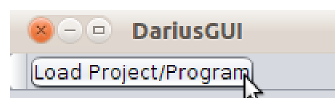


Figure 6.3: Button to load a Project/Program

Now that a program or software project if fully loaded, there is the possibility of exploring the other two components through the graphical interface of **Darius**. These two components can be

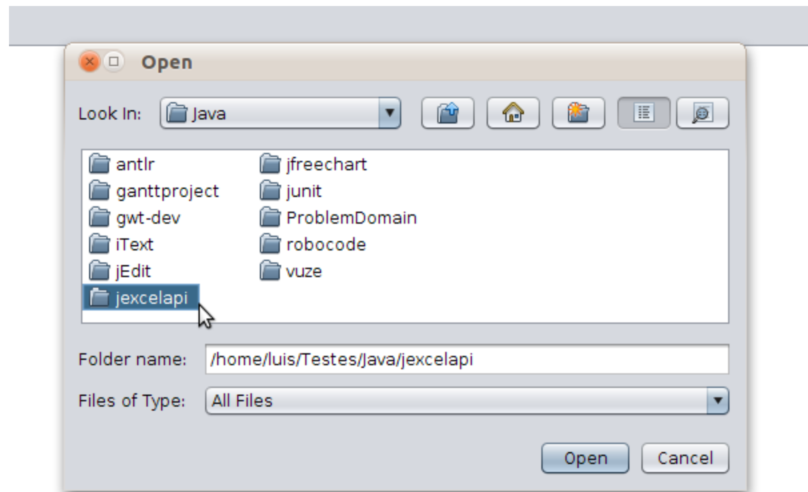


Figure 6.4: Loading a Project/Program

selected using the main tabbed pane on the center stage of the graphical interface. If the user chooses the Comments Statistics calculator, the **Darius** graphical interface will present the same result as it can be seen on Figure 6.5. If the user clicks on the *Load Statistics* button, **Darius** will then explore the program loaded and calculate statistics regarding the commenting practice on this program. In the end, the text area, on the center stage of this component, will be updated, presenting the results calculated by **Darius**, as it can be seen on Figure 6.6.

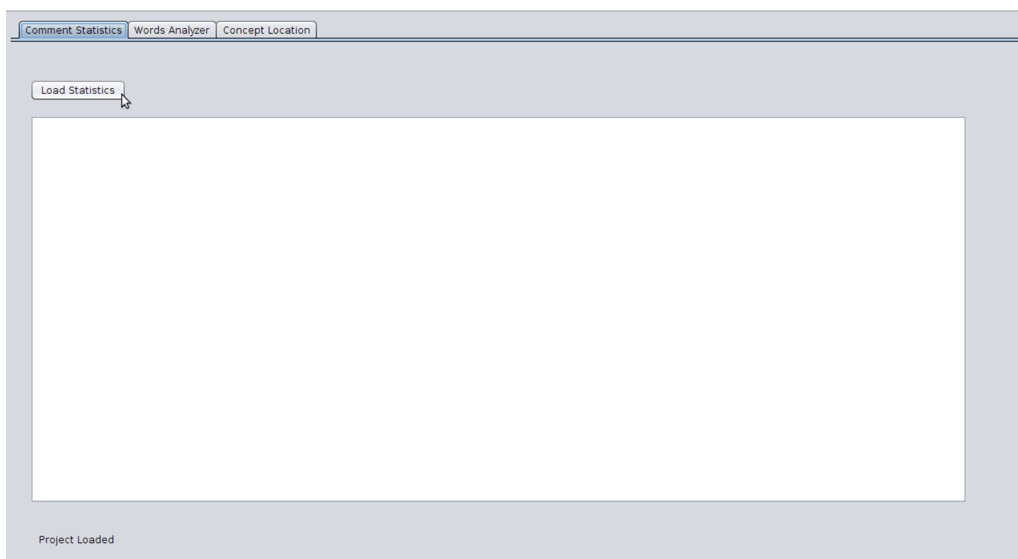


Figure 6.5: Comments Statistics Calculator Graphical Interface

Selecting the tab corresponding to the Comment Words Analyzer, the **Darius** graphical interface

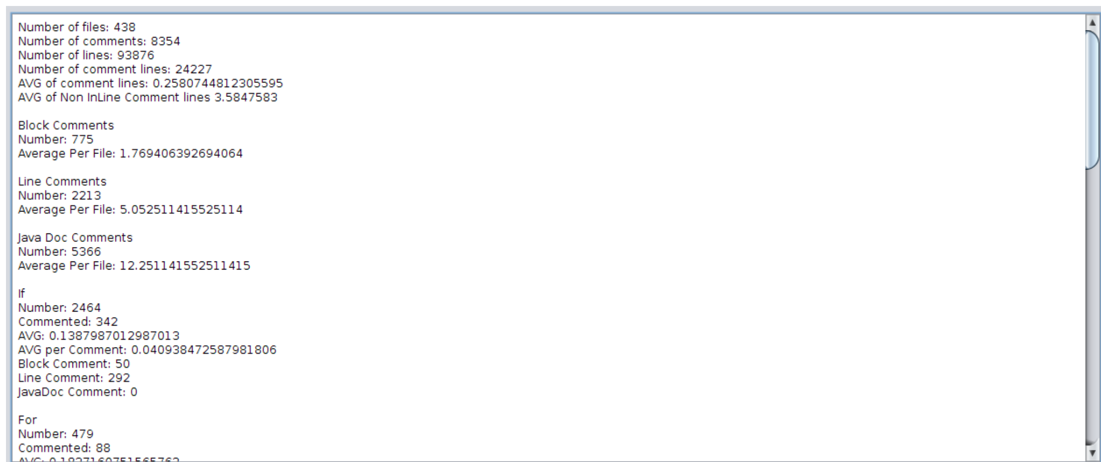


Figure 6.6: Results from the Comments Statistics Calculator

will present the graphical design of this component as it can be seen on Figure 6.7. In this graphical design, there is a text area on the left side of the center stage, where the user can collocate a list of words, that he wishes to be analyzed by this component, as it can be seen on Figure 6.8. After entering the desired words, the user can now click on the *Analyze* button, and **Darius** will then analyze these words in relation to the comments of the loaded program, using the **Darius** Comment Words Analyzer. The results from this analysis will be then presented on the text area on the right side of the center stage, as it can be seen on Figure 6.9.

6.3 Tests: Preliminary Study

As the development of the first version of **Darius**, and its graphical interface, was conluded it was possible to proceed for the objective of its development: executing the preliminary study that could prove the feasibility of developing a **PC** tool that uses comment information. The strategy and goals of this preliminary were already introduced on Section 6.1. The first step was choose the programs or software projects that would serve as objects of study. This choice and the reasons for it are presented on Subsection 6.3.1. Then, with the set of objects of study completely established, it was possible to proceed for the execution of the preliminary study. This was divided into two different tests: comment quantity and comment content. The results for the first test are presented on Subsection 6.3.2 and for the second one are presented on Subsection 6.3.3.

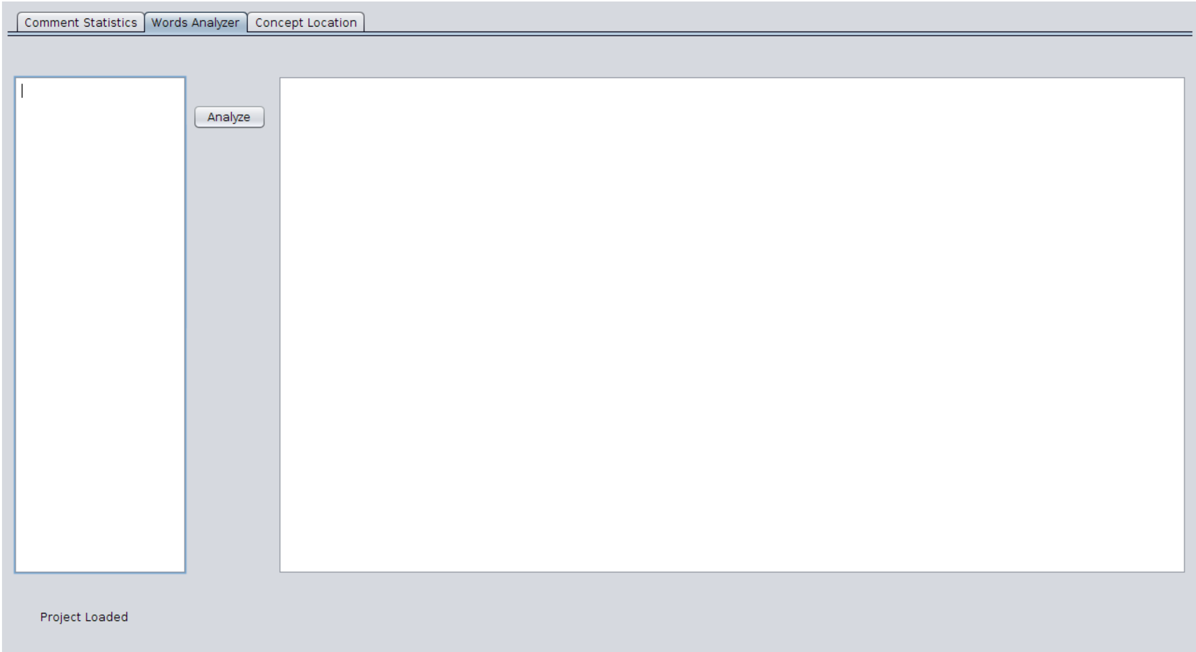


Figure 6.7: Comments Words Analyzer Graphical Interface

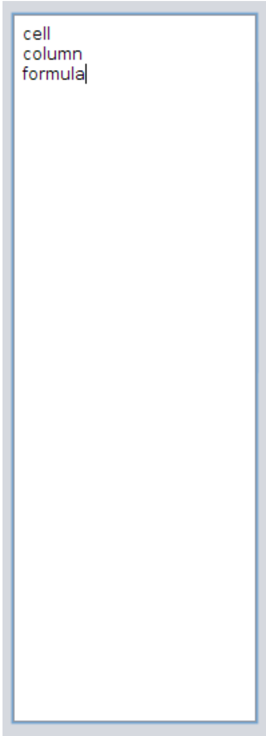


Figure 6.8: List of words to analyze

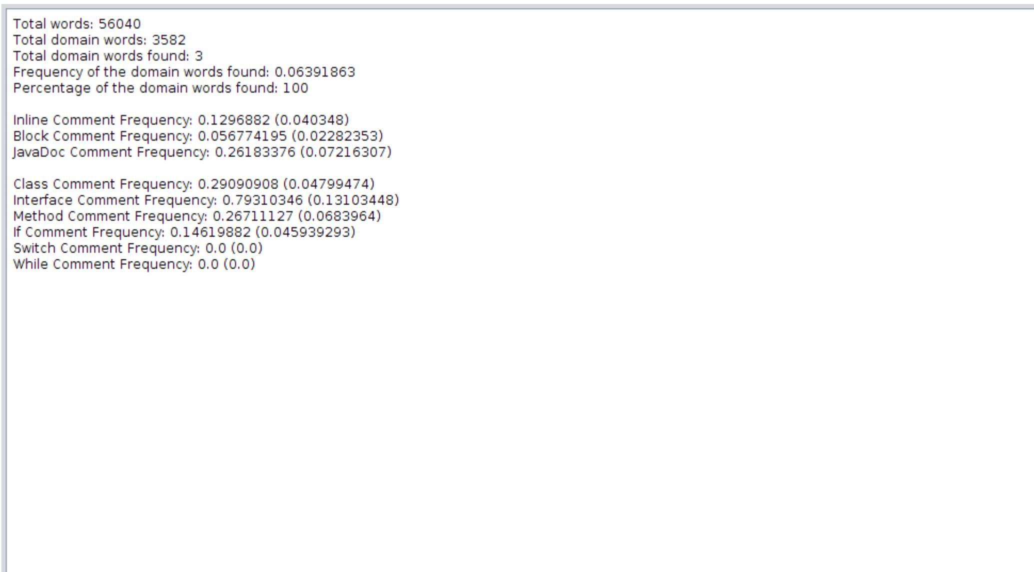


Figure 6.9: Results from the Comments Words Analyzer

6.3.1 Objects of Study

In order to perform this preliminary study, it was selected 10 software projects[3] written in the Java programming language. In this case, all of these 10 software projects are open-source. The choice for the use of open-source projects had two motives: the first is that the source code is totally free; the second motive is that open-source software projects are highly used by the community to change and manipulate the source code over and over again, and so, the comprehension tasks that these projects need tend to be more necessary, and so commenting can be a proper way of helping on these tasks. The selected projects are described on Table 6.1. The selection of these particular software projects has not followed any particular criteria, apart from the constraint that their Problem Domains should be different.

6.3.2 Comment Quantity

As was previously mentioned, the first test of the preliminary study is about the quantity of comments on the set of programs or software projects presented on the Subsection 6.3.1. The strategy applied in this test is fully described on Subsection 6.1.1. In resume, this was based on the calculation of the ratio between the number of comment lines and the number of source code lines. According

[3]These projects were retrieved from the SourceForge repository—http://sourceforge.net—in December 2010.

Project	Description	Files	LoC	Classes
iText	PDF Library	480	145666	403
ganttproject	Project Management Library	530	68945	394
gwt-dev	Google's Web Toolkit	987	192738	803
jEdit	Text Editor	531	176006	404
vuze	Peer-to-peer client	3284	785935	2463
junit	Tests Framework	154	10926	130
jfreechart	Chart Library	989	313231	876
antlr	Grammar Framework	221	85867	212
jexcelapi	Excel Library	438	93876	166
robocode	Programming Game of Robots	571	81519	485
Total		8185	1954709	6336

Table 6.1: Description and size of each selected project

to what is defined on Subsection 6.1.1, a program or software project must contain at least a ratio of 19% between comments and source code. So in order to explore the comments of a given program to enhance **PC**, they must be at least 19% of the whole program. However, and taking in consideration what is mentioned on Section 6.2.2, that this would be a very simple task, it was introduced several other statistical functions onto the **Darius** Statistics Calculator, that would provide important information about the commenting practice on a program, information that could be used for other purposes, apart from the **PC** purposes.

So, in order to execute this first test, each of the programs or software projects included on the set of elements of study were loaded into **Darius**, and the Statistics Calculator was able to calculate all of the comment statistics that it provides. The individual and global results of this first study of comment quantity are presented on Table 6.2.

On Table 6.2 it can be seen the total number of comments (#CM), inline (#IC), block (#BC) and javadoc (#JD); the total number of comment lines (#LOCM), the average of comment lines per comment (LOCMPCM) and the average of comment lines per lines of source code (LOCMCPC), for each project and the global results. The last value is the focus of this first test of the preliminary study. As it can be seen on Table 6.2, only the software projects *iText*, *jEdit*, *jUnit*, *jFreeChart*, *jexcelapi* and *robocode* respect the condition of the 19% ratio between comment lines and source code lines. So, thinking about the preliminary study, only these projects, if they pass the second test of the preliminary study which can be seen on Subsection 6.3.3, can be considered to explore for **PC** purposes. Amongst these candidates, *iText*, *jexcelapi* and *jfreechart* are those who contain a higher frequency of comment lines, with a value higher than 24%, almost one quarter of the whole

					Type of Comments		
Project	#CM	#LOCM	LOCMPCM	LOCMCPC	#IC	#BC	#JD
iText	13343	35246	3.6	0.24	4930	3777	4636
ganttproject	4468	7544	2.99	0.11	2925	814	729
gwt-dev	12969	31648	4.25	0.16	7219	866	4884
jEdit	18986	37749	5.11	0.21	806	14421	3759
vuze	27723	64023	4.83	0.08	18245	2319	7159
junit	519	2281	4.99	0.21	2	77	440
jfreechart	22516	83934	4.86	0.27	6592	2530	13394
antlr	5292	11678	5.6	0.14	3903	1380	9
jexcelapi	8354	24227	3.58	0.26	2213	775	5366
robocode	5071	15657	6.39	0.19	3108	102	1861
Total	119241	313987	4.5	0.16	63633	13371	42237

Table 6.2: Comments Frequency in the projects (detailed analysis).

program.

Now that the main conclusions for the first test of the preliminary study were obtained, it is now time to look for the other information extracted by this first test, that do not affect directly this preliminary study, but can be important for other purposes. This can be seen on Tables 6.2, 6.3 and 6.4.

Looking at Table 6.2 it can be seen, that in average, the programmers from the software projects in study use 4.5 lines to create a comment. It is important to note, that this value does not include the introduction of inline comments, due to the fact that it only contains one line by nature. So the conclusion is that block and javadoc comments include in average 4.5 lines of comment in the set of objects of study. On Table 6.2 it can also be seen the total number of the different types of comments. Without any doubt, inline comments are the most used type of comment, followed by javadoc comments and in the end block comments.

Table 6.3 shows the percentage of source code entities presented on the source code of the set of elements of study, that are effectively commented. The results provide interesting but rather expected results. Classes, interfaces and methods are the most type of source code entities which are commented, with a large difference of percentage in relation to the rest of source code entities. This shows that programmers in this set of software projects, tend to comment more source code entities with a higher level of abstraction. However, this can also be due to the fact that these source entities have a less number of unities on the source code in relation to the others, and so they are more easy to comment. Another important and interesting point to extract from this table is the

Project	If	For	While	Switch	Class	Interface	Method
iText	5	7	7	7	89	90	76
ganttproject	5	5	3	8	57	41	18
gwt-dev	9	10	7	5	96	97	19
jEdit	9	8	4	2	86	79	61
vuze	6	6	5	7	45	46	24
junit	1	0	0	0	25	71	37
jfreechart	6	10	2	18	100	100	100
antlr	11	16	5	4	61	56	22
jexcelapi	14	18	12	0	99	100	88
robocode	7	11	12	3	76	94	20
Total	7	8	6	5	69	60	45

Table 6.3: Percentage of Source Code Entities (SC) commented (#SC commented / #SC)

Project	If	For	While	Switch	Class	Interface	Method
iText	IC	IC	IC	IC	JD	JD	JD
ganttproject	IC	IC	IC	IC	JD	JD	JD
gwt-dev	IC	IC	IC	IC	JD	JD	JD
jEdit	IC	IC	IC	IC	JD	JD	JD
vuze	IC	IC	IC	IC	JD	JD	JD
junit	IC	N/A	N/A	N/A	JD	JD	JD
jfreechart	IC	IC	IC	IC	JD	JD	JD
antlr	IC	IC	IC	IC	BC	BC	BC
jexcelapi	IC	IC	BC	N/A	JD	JD	JD
robocode	IC	IC	IC	IC	JD	JD	JD
Total	IC	IC	IC	IC	JD	JD	JD

Table 6.4: Most used type of comment per type of source code entity

fact that in *jfreechart*, all of the classes, interfaces and methods presented on its source code, are commented.

To conclude, Table 6.4 shows the relation between a type of source code entity commented and the most used type of comment used to comment it. The results show again another high contrast between high level source code entities and lower level source code entities. In average, the first group is mostly commented using javadoc comments, unlike the second group were the inline comment is mostly used. With this, it can be concluded that Javadoc comments are mostly for the use of high level commenting, leaving inline comments for the lower level commenting.

6.3.3 Comment Content

The second and final test of the preliminary study, that pretended to study the feasibility of creating a tool that uses comment information to enhance **PC**, was about exploring the content of comments, by checking whether it is used Problem and Program domain information, on the set of elements of study presented on the Subsection 6.3.1. The stategy applied in this test is fully described on Subsection 6.1.2. In this case, the objective was to measure the number of words from the Problem and Program domain, enumerated on lists, that occur on the comments. The goal to succeed in this test, is that about 23% of the words from the list must be included on the comments, being this value a reference as it is explained on Subsection 6.1.2. Although this calculation would be sufficient to conclude this test, as it is mentioned on Section 6.2.3, this would be a very simple task, so it was introduced other functions to the **Darius** Comment Words Analyzer, that provide interesting information regarding the use of Program and Problem Domain contents on comments.

To execute this test, it was created a list of Program Domain terms for each of the software projects presented on the set of elements of study, by checking the websites and manuals, looking for terms on the requirements that are contextualized on the Problem domain. Each of the lists and the respective program were submitted to the **Darius** Comment Words Analyzer, which returned the values of the analysis. Then it was created a single list of Program Domain terms, just one, because the programming language is the same for every project. Then the same routine was executed: this list and each of the projects were submitted to the **Darius** Comment Words Analyzer, which returned the values of the analysis. The results of all the analysis can be seen on Tables 6.5, 6.6, 6.7, 6.9, 6.8 and 6.10.

Table 6.5 shows the percentage of words from the lists that occurred on the comments of the software projects, which was the objective of this second test of the preliminary test. Looking at the table, it can be seen that every project has a percentage much more higher than the value of reference, 23%, so they all pass this test. Taking in consideration the results obtained and explained on Subsection 6.3.2, where only *iText*, *jEdit*, *jUnit*, *jFreeChart*, *jexcelapi* and *robocode* passed on the first test, only these candidates passed this preliminary study, as they also passed the second test. So, in conclusion, these software projects proved the feasibility of the development of a Comment Analysis **PC** tool, and proved themselves as possibles subjects for the execution of tests of that same tool.

Apart from the main conclusions that were the focus of this second test of the preliminary

Project	Problem Domain	Program Domain
iText	92.31	86.76
ganttproject	84.31	75.0
gwt-dev	56.34	86.76
jEdit	89.74	86.76
vuze	92.11	88.24
junit	81.82	67.65
jfreechart	86.36	89.71
antlr	88.24	83.82
jexcelapi	79.31	85.29
robocode	88.89	83.82
Total	82.21	83.38

Table 6.5: Percentage of domain words (DW) found (#DW Found / #DW)

study, there were other values regarding the use of Problem and Program Domain information on comments, that **Darius** Comment Words Analyzer was able to calculate. These can be seen on 6.6, 6.7, 6.9, 6.8 and 6.10.

Table 6.6 shows the frequency of words from the lists on the comments. As it can be seen on the table, in average, 11.61% of the total number of words included on comments are Problem Domain oriented. However, this percentage is even higher in the case of Program Domain terms, obtaining a 17.71% of the total of words. In total, almost 30% of the total terms presented on comments, are Problem or Program Domain oriented. The differentiation of these values between the type of comments and source code entities can be seen on Tables 6.7 and 6.9. Table 6.7 shows the frequency of words from the lists on the three types of comments. As it can be seen the content of JavaDoc comments is the most oriented for Problem and Program Domain information, as it contains the higher frequency for both domains. The block comments, on the other side, are the type of comments with the least frequency of information, also on both domains. These results are sustained by the values presented on Table 6.8, that shows the percentage of types of comments that contain Problem and Program Domain information. As it can be seen, in average, almost 74% of the javadoc comments contain Program Domain terms and 48% of them contain Problem Domain terms. Once again, the block comment has the worst results in terms of Domain information. On terms of Problem Domain words frequency, it has less than a half compared to the javadoc comment, and on term of Program Domain the same ratio maintains.

In terms of source code entities, Table 6.9 shows the frequency of words from the lists on the different types of source code entities commented. As it can be seen on the table, on terms of

Project	Problem Domain	Program Domain
iText	13.39	16.21
ganttproject	14.51	12.77
gwt-dev	2.27	20.99
jEdit	8.09	19.17
vuze	4.68	15.72
junit	22.5	25.52
jfreechart	15.89	20.62
antlr	13.84	12.78
jexcelapi	22.75	16.37
robocode	23.39	12.54
Total	11.61	17.71

Table 6.6: Frequency (%) of words of each Domain (#Total DW / #Total Words)

	IC		BC		JD	
Project	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**
iText	13.05	13.99	6.89	14.1	14.7	17.36
ganttproject	13.76	13.52	14.78	11.58	14.67	14.2
gwt-dev	0.96	19.7	2.03	18.31	2.62	22.16
jEdit	5.1	17.15	6.44	24.76	9.28	16.69
vuze	4.6	18.02	5.14	11.38	4.29	18.89
junit	0	20.0	17.14	16.57	22.66	25.77
jfreechart	20.7	20.73	16.74	12.45	15.58	21.41
antlr	13.85	13.81	13.95	10.7	2.13	11.35
jexcelapi	10.38	16.16	17.08	12.97	24.97	17.01
robocode	17.0	14.06	16.52	12.6	25.13	12.5
Total	9.97	16.27	8.33	14.58	13.13	19.13

Table 6.7: Frequency (%) of words of each Domain per type of comment

	BC		IC		JD	
Project	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**
iText	33.94	35.5	12.45	24.65	59.17	72.11
ganttproject	45.58	43.49	29.4	30.32	45.27	50.89
gwt-dev	7.85	71.13	5.25	37.37	19.68	78.05
jEdit	25.56	50.5	11.86	42.98	54.8	72.07
vuze	16.65	46.44	15.57	32.47	21.71	61.03
junit	0	50.0	27.27	29.87	77.73	82.95
jfreechart	48.18	48.97	33.16	25.94	56.87	86.23
antlr	66.16	71.96	31.9	30.0	11.11	33.33
jexcelapi	20.0	42.45	43.97	35.25	66.05	65.58
robocode	60.78	32.35	22.27	23.94	51.26	58.89
Total	34.87	47.79	18.11	33.55	47.6	73.72

Table 6.8: Percentage of each type of comments that contains the type of Domain words

frequency of Problem and Program Domain information, methods have the highest values. Almost one third of the content on methods comments is Program or Problem Domain oriented. Apart from the exception of methods, lower-level source code entities like ifs, fors, whiles and switches have the highest frequency of Program Domain information. On contrast, methods, classes and interfaces have the highest frequency of Problem Domain words. So with these two facts, it can be created a relation between the level of abstraction of the source code entity and the nature of the content on the respective comment: higher level source entities tend to have comments oriented for Problem Domain information, whereas comments of lower level source entities tend to include more Program Domain information. Table 6.10 shows the percentage of source code entities commented that contains at least one word from the Problem and Program Domain list. From this table, as it can be seen, it is sustained the the idea of the richness of Program and Problem Domain information on the comments of methods. Another interesting fact extracted from this table, is that all of the comments of interfaces on *junit* contain at least one word from the Problem Domain and one word from the Program Domain.

	If		For		While		Switch		Class		Interface		Method	
Proj.	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**
iText	8.27	16.45	6.5	16.61	5.26	14.47	3.13	12.5	17.49	11.69	18.13	14.23	13.54	17.53
ganttproject	19.0	9.26	20.61	14.5	30.0	15.0	0	50.0	11.82	9.87	5.66	8.02	15.35	17.14
gwt-dev	0.99	19.99	0.25	18.48	2.5	11.67	0	33.96	3.57	18.33	3.34	16.02	2.33	23.32
jEdit	5.05	18.15	4.69	12.02	3.92	9.8	3.13	6.25	9.32	11.31	9.54	13.21	10.09	19.8
vuze	5.59	12.58	4.49	13.29	3.8	14.45	1.43	17.14	3.09	9.02	4.82	11.3	4.11	21.08
junit	20.0	20.0	0	0	0	0	0	0	24.35	17.41	19.05	23.81	23.77	27.14
jfreechart	16.18	16.46	19.33	14.71	5.0	15.0	45.45	0	17.19	16.84	15.5	18.64	15.05	22.46
antlr	13.54	13.7	8.93	12.39	9.26	14.81	0	32.61	14.84	10.97	13.03	14.79	13.78	14.26
jexcelapi	15.75	13.7	20.9	12.54	4.82	15.66	0	0	24.92	12.69	28.28	25.86	23.62	18.87
robocode	14.12	14.24	8.42	23.16	17.39	21.74	25.0	25.0	11.99	5.6	15.66	6.05	30.04	14.65
Total	7.38	15.1	7.19	14.99	5.87	14.32	6.04	22.63	11.1	12.63	9.58	13.62	13.24	20.51

Table 6.9: Frequency (%) of words of each Domain per source code entity

	If		**For**		**While**		**Switch**		**Class**		**Interface**		**Method**	
Proj.	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**	**Prob.**	**Prog.**
iText	21.39	40.06	17.44	34.88	11.54	30.77	5.88	23.53	78.06	60.83	81.08	78.38	50.18	66.43
ganttproject	38.46	23.08	51.72	44.83	40.0	60.0	0	66.67	31.39	37.22	23.81	14.29	46.75	58.61
gwt-dev	4.58	55.99	1.2	47.31	12.5	29.17	0	83.33	28.33	73.74	28.57	72.11	18.0	79.32
jEdit	16.69	46.68	17.86	36.9	11.76	29.41	14.29	28.57	63.04	59.03	75.0	68.75	43.18	82.14
vuze	18.99	38.72	14.05	32.44	11.84	32.89	8.33	25.0	13.58	26.54	18.39	26.0	22.15	67.73
junit	50.0	50.0	0	0	0	0	0	0	87.5	87.5	100.0	100.0	75.83	79.46
jfreechart	41.76	44.64	52.38	39.68	20.0	40.0	100.0	0	72.46	85.6	77.67	94.17	53.29	87.29
antlr	38.08	50.12	29.41	36.76	30.77	23.08	0	100.0	71.54	65.38	90.0	80.0	54.76	60.53
jexcelapi	40.35	38.01	53.41	38.64	13.04	43.48	0	0	89.7	60.61	96.55	79.31	64.51	77.57
robocode	34.65	50.5	23.91	50.0	43.75	43.75	100.0	100.0	33.61	26.23	31.94	37.5	53.81	66.84
Total	22.7	43.57	21.51	38.39	16.59	34.15	11.43	51.43	44.81	55.4	35.97	47.09	45.76	77.35

Table 6.10: Percentage of each source code entity comment that contains the type of Domain words

6.3.4 Discussion of the results

The objective for the development of the first version of **Darius** was to prove the feasibility of creating a second version of it, that incorporated the exploring of comment information for program understandability purposes. With **Darius** first version, it was possible to execute the preliminary study that would prove that feasibility, by performing two different tests on a set of programs or software programs.

The first test focused on the ratio between comment lines and source code lines, that would determine the commenting percentage on a given program. If programs do not have a sufficient number of comments, it is not worthy to explore the comment side of them to try to understand it. Having a value of reference for that ratio, the results showed that from that set of 10 software projects, only six of them had a higher or at least the same amount of comments percentage: *iText*, *jEdit*, *jUnit*, *jFreeChart*, *jexcelapi* and *robocode*. However, this was sufficient to claim that this first test of comment quantity have obtained success, so Question 2 from Section 1.2, has been answered positively: *there are programs like iText, jEdit, jUnit, jFreeChart, jexcelapi and robocode, that have a sufficient amount of comments, taking in consideration a value of reference.*

Having passed the first test, the focus was in the moment on the second test of the preliminary study. The objective of the second test was to analyze the contents of the comments of the set of software projects. In particular, this content analysis checked the use of Program and Problem Domain information on comments, on the form of words. The strategy followed to conclude this test was to create lists of words for the both types of domains and check which percentage of those words were included on the comments. Having also a value of reference for this percentage, the results showed that all of the projects contained a higher value for that reference, so they all passed this test. With this fact, it is now possible to answer positively to Question 3 from Section 1.2: *there are programs like iText, jEdit, jUnit, jFreeChart, jexcelapi and robocode, that contain comments that include problem and program domain information, having a value of reference for the percentage of that information*.

Having Questions 2 and 3 answered positively, 1 from Section 1.2 is automatically answered as positive: *it is worthwhile to use comments to develop a tool that explores comment information to enhance PC*. With this, it was possible to advance for the development of the second version of **Darius**. The programs of software projects that pass the preliminary study were apt to be candidates for subjects for this second version, and maybe be the proof for the success of this tool.

6.4 Summary

In order to proceed for the development of a Comment Analysis **PC** tool called **Darius**, it is necessary to study the nature of comments, to see if they are worth to explore for **PC** purposes. To do so, it was developed a preliminary study that explored two important factors that are directly affected with the ability of comments as keys for comprehension: their quantity and their content.

To study the amount of comments or the percentage of commenting on the source code, the strategy was to calculate the ratio between comment lines and source code lines, from a set of objects of study, and compared it to a value of reference. If some program had at least that value of ratio, it was able to pass this test, and the preliminary study could proceed for the second test.

In the case of the study of the content of comments, this was based on *Brooks* statement, that comments help on the comprehension if they provide Problem and Program Domain information and means to establish bridges between those two domains. To execute this test, the strategy was to calculate the percentage of words from lists (with words from the Problem and Program Domain) that occurred on the comments, and compared it to a value of reference.

In order to execute this preliminary study with more efficiency and pace, the decision was to create a first version of **Darius** that included modules that provided means of executing the two different tests. For each of the tests it was created a different module and a graphical interface in order for the visualizing of the results to be more effective.

The first module or component created for **Darius** was the comment extractor, that extracted comments from *Java* source code files using regular expressions. The next line of source code line was also extracted and associated with the comment. The second module was the comment statistic calculator that calculated the ratio of comment line per source code line, but also calculated other important values regarding the frequency of comments on a given program. The third module was the comment words analyzer that received a list of words and calculated the percentage of those words that appeared on comments. Apart from just calculating this percentage, the words analyzer included other important functions.

The first test of the preliminary study, who focused on the quantity of comments, showed that six of the ten programs in test had the right amount of comments taking in consideration the value of reference. From the other measures returned by **Darius**, important information were also able to be extracted regarding the nature of the frequency of comments on the programs. In particular,

a non inline comment has in average 4.5 lines and the javadoc comment is the most used type of comment. It was also able to conclude that classes, interfaces and methods tend to be more commented, with a high difference to the others source code entities, and that these source code entities are mostly commented by javadoc comments, whereas the others are mostly commented by inline comments.

The second test of the preliminary study focused on the calculus of the percentage of words from the Problem and Program Domain, inserted on a list, from the comments, showed that all the ten programs contained a percentage higher that the value of reference. Other important conclusions were also extracted from this study. In particular, almost 30% of the contents of comments are Problem or Program domain oriented, on terms of words, but there is a higher frequency of Problem Domain words. Javadoc comments is the most rich type of comment on terms of Domain information. From these results, it was also able to make a relation between the frequency of the different type of source code entities commented and the type of knowledge domain inserted on comments: in general, lower-level source entities include more Program Domain information on their comments, whereas higher-level ones include more Problem Domain information.

The results from the two tests have shown that the preliminary study was a success, and that Question 1 was answered positively. There was conditions to proceed for the development of a Comment Analysis **PC** tool.

Chapter 7

Darius: The second stage

Having Questions 2 and 3 from Section 1.2 been answered positively, and so forward, Question 1, it was safe to move forward for the development of an approach that takes advantage of the source code comments power to enhance **PC**, to try to answer to Question 4. This question raises the problem of using source code comment information to enhance the comprehension of a program, by providing it to the user in order for him to establish a bridge between the Problem and Program Domain. This problem was already addressed by several authors, as it is mentioned throughout Chapter 3. In that chapter are fully detailed several approaches that try to address this problematic, by introducing **IR** techniques, and adapting them to the use of source code information, in order to search for Problem Domain concepts on the source code, and in this way, providing means to the programmer to establish a bridge between the Problem Domain and the Program Domain, that would help him understand a program faster and with more efficiency.

Due to the multiple and variety of approaches of this kind that provided so many good results, the best and wiser decision was to address this problem in the same way, by using the same strategies and methods. Although there was the alternative of taking the direction of an approach similar to the one described on Section 5.2 that provided quality results, the development of this approach would be very hard and would take a lot of effort and time, as this would require the development of a knowledge base, an inference system and a natural language parser, amongst other components of high complexity. To this is added the fact that it is not proven that this approach provides better results than the **IR** based ones, as in the literature, the second type of approach continues to be explored and improved, as opposed to first type, that was not sustained, as the last reference to it is from 2008.

So the next step was to create a second version of **Darius**, that would prepare it to explore comments for the searching of Problem Domain concepts and to give possibility to find the maps of these concepts on the source code. The development of this approach included the use of **IR** methods and techniques to make this search possible. All of the information regarding these techniques are fully available throughout Chapter 3. With these, the strategy was to adapt the idea behind **IR** to comments, and create a search engine that would be prepared to answer to queries, that correspond to the programmer needs of information regarding understandability. In this case, it would be considered that with this tool, the user is looking for a way of seeing what piece code corresponds to the materialization of a certain Problem Domain concept. For instance, if when dealing with an unfamiliar software of store management, the programmer wants to change the layout that the program uses to print the receipt of a sale, the programmer introduce in **Darius** a query corresponding to that need, and **Darius** would provide the code, taking in consideration the comments used, that answer to that need.

The strategy followed to address this problem is very similar to most of the **IR** for PC approaches, that can be seen on Section 3.2

In this chapter the focus is on the second version of **Darius**, as having the capacity of enhancing **PC**. The chapter starts with Section 7.1 that gives a detailed description of every module developed for this second version, as well as the graphical interface, indicating and substantiating the decisions that were made on the course of the development. Section 7.2 ends this chapter, presenting a rigorous but enlightening test that was executed to explore the capacity of **Darius** second version, as a **PC** tool. It includes all of the obtained results and also the proper discussion and the conclusions extracted from those results.

7.1 Darius: Version Two

As was mentioned before, in order to address the *concept assignment problem*, this second version of **Darius**, followed the same strategies followed by approaches of similar goals and purposes and that are discussed throughout Chapter 3. In order to do so, this second version included the following components (Figure 7.1):

- the logical view of the documents, or more simple, the implementation of the document database, that stores all of the information that is included on the contents of the comments;

- **IR** models that explore the information of the document database, and retrieve documents according to the queries the users provide;
- a graphical interface that provides good user experience for a higher efficiency on the exploration of the process of finding Problem Domain concepts using comment information.

The following subsections described in detail each of the components mentioned above.

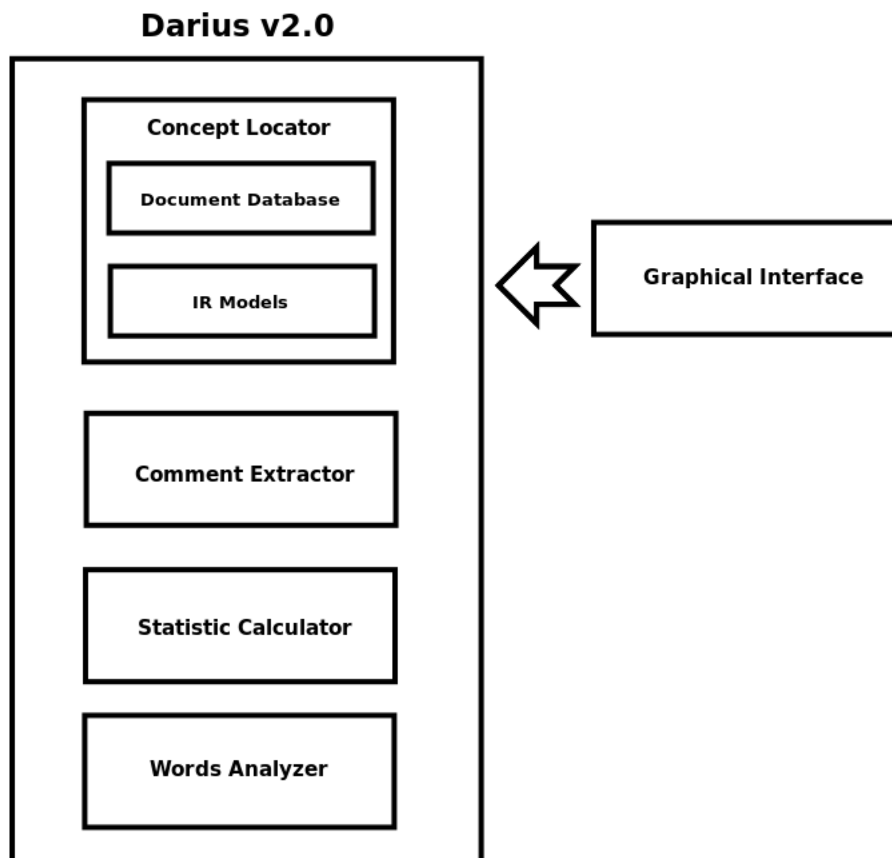


Figure 7.1: Darius Second Version Structure

7.1.1 Document Database or Corpus Implementation

So the first step was to create the document database. In this case the decision was to choose granularity at comment level, so each comment of a program will be considered a document in this database or corpus. The reason for this decision comes from the fact that source files can have a big size and can correspond to several Problem Domain concepts on code, and a granularity of this level would turn very difficult the search of some specific and less generalized concept. So considering

granularity at comment level could provide to the user quick and straightforward answers to the questions he has in reality.

Established the decision of considering granularity at comment level, it was time to focus on the extraction and manipulation of the words that were going to be part of the *inverted index* (Section 3.1) of the document database. The first step in this phase was to extract the words from the comments. This is accomplished using an object of the class *JavaCommentsWords*, which of whom is fully detailed on Section 6.2.3. Having the complete set of words from the comments is was possible to build the document database. To represent it, **Darius** uses a matrix words $\times$ comments, which defines the weight of a certain word on a given comment. To do so, **Darius** contains a class called *WordCommentMatrix*, whose object receives a list of comments and a set of words, and builds a matrix, array of arrays in this case, which in every position colocates the frequency of the word, correspondent to the line of that position, on the comment, correspondent to the column of that position. In the end, **Darius** will contain the complete matrix that represents the document database. However the process is not yet complete. It was also necessary to redefine the weights of the words on the documents, apart from the simple frequency. To do so it was applied the **TF-IDF** algorithm, whose description is fully available on Subsection 3.1.1, and is also mostly used in **IR**. With the application of this algorithm, the matrix would represent better the real importance of words on the comments extracted.

7.1.2 Vector Space Model Implementation

With the document database fully defined, it was possible to build **IR** models that would explore that database, and retrieve documents, or comments in this case, that correspond better for a given query. The first and most simple implemented **IR** model was the **VSM**. The full description of this model can be seen on Subsection 3.1.2. In **Darius**, the implementation of **VSM** is defined in a class called *VSM*. To build an object of this class it is necessary to pass a *WordCommentMatrix* object, that corresponds to the document database, as was seen on the previous section. Apart from this, and taking in consideration the description of this model, there was nothing more to do apart from doing queries to this model. To do so the **VSM** class contains a method that provides the mean to execute a given query to this model. This method receives a list of words or *strings* that correspond to the query of the user. This query is then defined as an array that has the size of the total number of words of the inverted index of the document database. To build this array,

Darius collocates the number one on the correct position for each of the words of the passed query. Then it is necessary to execute this query. To do so, and as defined on Subsection 3.1.2, **VSM** uses the *cosine similarity* to rank the query towards the documents or, in this case, comments. The mathematical implementation of this algorithm is also described on Subsection 3.1.2. This will be calculated between the query and each one of the comments, using the columns of the matrix, previously build, and will return a value between 0 and 1. According to **VSM**, the value of the *cosine similarity* is directly proportional to the similarity of the comment to the query.

7.1.3 Latent Semantic Analysis Implementation

Having the **VSM** totally defined in **Darius**, it was necessary to implement a more complex **IR** model, which in this case would be the **LSA**. The description of this model and its implementation can be seen on Subsection 3.1.3. In **Darius**, the implementation of **LSA** is defined as a class called *LSA*. As well as the class *VSM*, detailed on the previous section, the construction of an object of the *LSA* class, needs the passage of the document database or corpus in the form of a *WordCommentMatrix* object. As it is seen on Subsection 3.1.3, **LSA** relies on the use of the **SVD** to factorize the matrix and reduce its dimensions. As was seen before, this will permit a better approximation of words and documents (comments) that are semantically connected. However, taking in consideration the complexity of the development of a **SVD** algorithm, the decision was to incorporate the use of a third-party **SVD** algorithm, which in this case is a **SVD** algorithm included on the *S-Space Package*[1]. This algorithm takes a matrix, which in this case is the words $\times$ comments matrix, and factorizes it into three new matrices as defined on the Equation (3.4) on Subsection 3.1.3. However, this algorithm needs the attribution of the order number or was seen before, the number of *concepts*, that provides the best approximation to the original matrix, and establishes the best semantical relationships. According to the authors in [58] the optimal number of dimensions or order is between 200 and 300, so the decision was to choose a value in between, that is 250 dimensions for **Darius**.

Having the model completely defined, in the end **Darius** will have three matrices as defined on Subsection 3.1.3. The first one corresponds to the position of every word on the new semantic space and the third one will have the comments positions. The second one will contain the single values. With this matrices, it is now possible to execute queries to the model. To execute a given

[1] S-Space Package, available on http://code.google.com/p/airhead-research/

query, this *LSA* class contains also a method that provides that function. It only needs a list of the words that take part of the query. And with it, this method ranks the query towards the comments, defining which ones are more relevant to the query. The first step in this case, is to represent this query into the new space that was build. To do so, **Darius** introduces an algorithm, fully detailed on [13], that represents the query on the new semantic space in order to perform appropriate ranking calculations. The mathematical model of this algorithm can be seen on Equation (7.1). q in this case corresponds to the array of the query, and U and S are the same matrices defined on Equation (3.4).

$$q = q^T U S^{-1} \tag{7.1}$$

With the query fully positioned in the new space, it is now possible to rank it in relation to the comments on the corpus. The same principle, as the one applied on the **VSM**, is used, as the cosine similarity is calculated to find out which of the comments are more related to the query defined by the user.

7.1.4 Graphical Interface

Having the implementation of the back-end for the second version of **Darius**, which includes the exploration of comment information for the enhancing of **PC**, through the search of Problem Domain concepts on the source code, it was necessary to create a graphical interface to support it, so that the users and programmers can increase the pace of its understandability tasks in a more effective manner.

Continuing the thought stated on Section 6.2.4, that there was not going to be put to many attention on extreme design details, but focusing only on the efficiency, the development of the graphical interface for the second version of **Darius** continued the basis of the first version. However in this second version, there was added an other tab onto the main tabbed pane, that corresponds to the new component, the concept locator, as it can be seen on Figure 7.2.

If a program or software project is correctly loaded, as described on Section 6.2.4, the user can now be able to start his concept location task. To do so, first he has to choose which **IR** model, that **Darius** provides, he wants to use. In this case, as it was seen before, **Darius** provides two

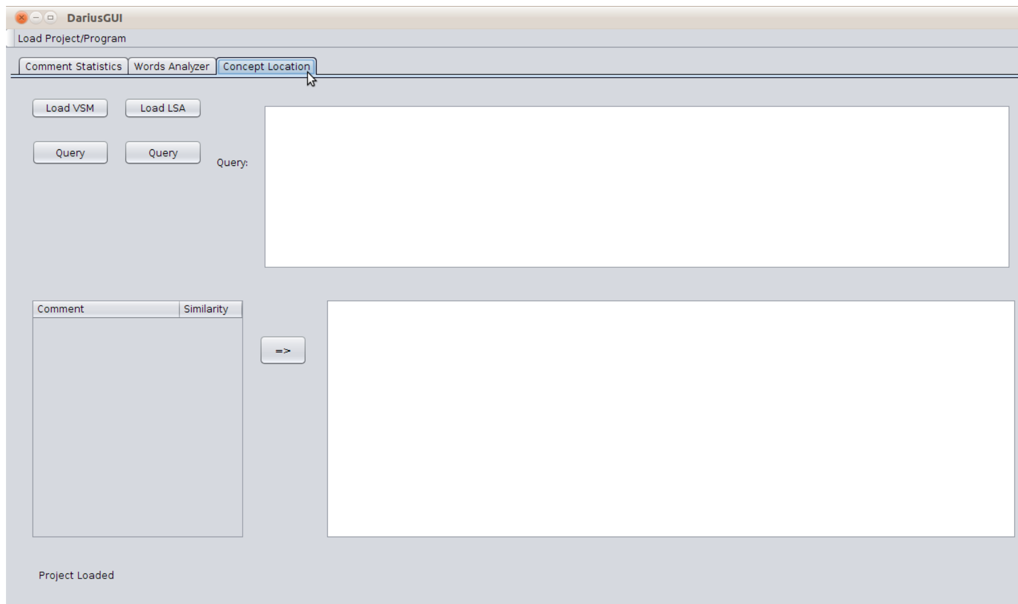


Figure 7.2: Concept Location Graphical Interface

different models: the **VSM** and the **LSA**. To do so, the user can click on the *Load VSM* button or on the *Load LSA* button, corresponding to model he desires, as it can be seen on Figure 7.3.

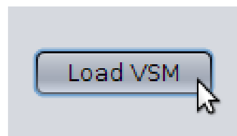


Figure 7.3: Loading an IR model

In the example of the Figure 7.3, the user clicked on the *Load VSM* button, that corresponds to the loading of the **VSM**. If everything runs correctly, in the end the model will be loaded, and **Darius** warns that fact by writing the message *VSModel loaded!* on the down side of the centre stage. Now the user can create a query and execute it in this model (Figure 7.4). To do so, on the top right side of the centre stage, there is a text area where the user can collocate loose words corresponding to the query he desires to be executed. Doing that, the user is now able to execute that query, by clicking on the *Query* button, which is under the button correspondent to the model he previously loaded.

After the query is executed, the result will be something like what is seen on Figure 7.5. The results of the query will be presented in a form of a list on the table which is on the left down side of the centre stage. This table will have two columns: the comment and the value for the similarity of

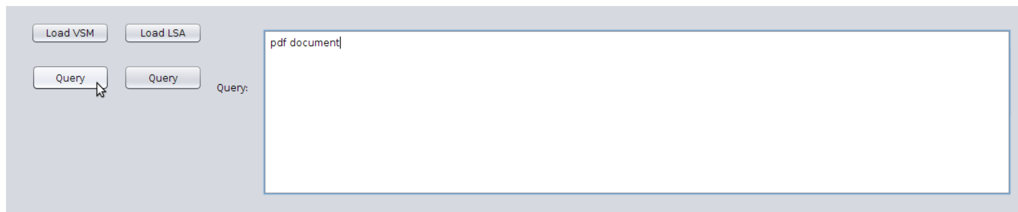


Figure 7.4: Writing a query to execute

that comment to the query executed. The results on the table will also be presented in a decreasing order of similarity value. According to the desires of the user, he can navigate throughout this list. select one of its element and see what is the type an nature of the comment he chose. To do so, he must select one element and click on the => button. By doing so, the text area on the right down size of the centre stage will be updated. In this text area it will appear the content of the comment, the source code associated with it, the file corresponding to that comment and the position of that comment on that file. With all of this information, the user can now be able to search for concepts on code on a more efficient way.

Figure 7.5: Results from a query

7.2 Test

In this section it is described a test which was executed in order to see the utility and efficiency of the second version of **Darius**. It is fully described, throughout this chapter, the functions that this version provides to the users. In resume, using **IR** techniques, **Darius** provides means to the users to search and locate Problem Domain concepts on the code, by using comment information presented on the source code. If the user is able to find concepts on the program, he can establish a bridge between the Problem and the Program Domain, which is a sign for a best and faster **PC**.

The object of study chosen to be subject on this test, is *iText*. This program took part of the preliminary study, described on Section 6.3, and passed it with success. According to the conclusions of that preliminary study, *iText* contains a sufficient amount of comments, and the contents of that comments have a sufficient dose of Problem and Program domain information, and so this program can be explored for **PC** purposes using its comments. Throughout this section, it is explained the goals and objectives of this test, as well as the results returned by **Darius**.

7.2.1 iText

As it was mentioned, the object of study for this test was *iText*[2]. *iText* is a Java library that allows to manipulate and create PDF files. The selection of this software project, apart from having passed the preliminary study, is due to the fact that the Problem Domain of this program is fully specified and its fully known by the generality of the programmers, who are fully aware of the PDF world.

Throughout the observation of the website and manual of *iText*, it was possible to extract most of the Problem Domain of this software project. The representation of the concepts from the Problem Domain and its relations can be seen on Figure 7.6. Like any other type of file, a PDF file has metadata that provides information from the contents of the file. In this case, and taking in consideration that is a PDF file, this metadata includes information regarding the author of the document, the attributed title, the date of the creation of the file, the subject, and a set of keywords that were given by the author to the document. A PDF document is a structure of objects than can be of different types: array, string, boolean, name, number and dictionary. All of these are low-level objects are then organized, creating special hierarchical structures that form higher-level structures called and defined by *iText* as elements, and that can be defined by the user. These include paragraphs, chunks, sections, phrases, paragraphs, anchors, tables, images, an lists, which are sequences of elements.

7.2.2 Concept Location

The Problem Domain of *iText* was defined, and the concepts around *iText* were established. With this information, it was possible to use **Darius** to search for these concepts on the source code, using the information presented on the comments. The strategy, to perform this test, passed by

[2] http://itextpdf.com/

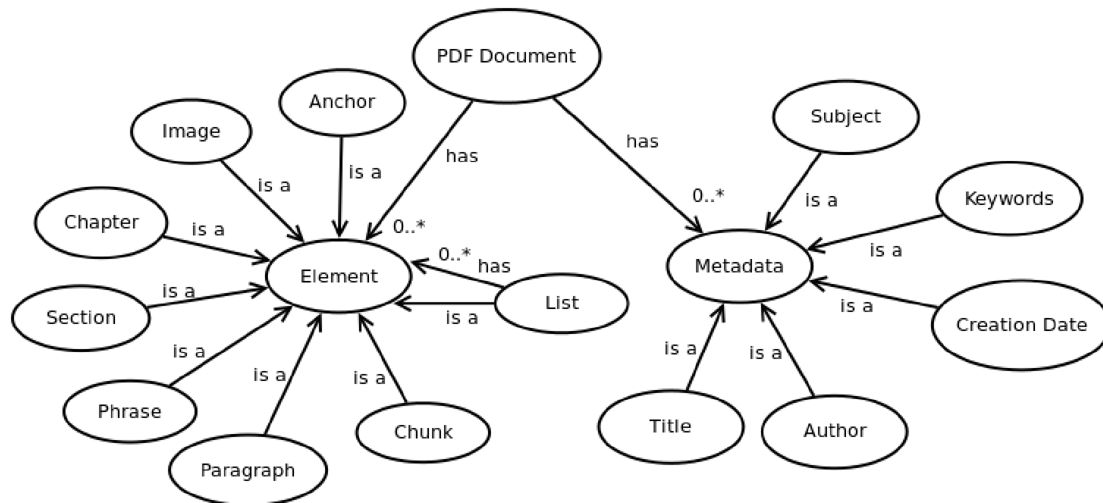


Figure 7.6: iText Problem domain

the execution of queries, whose nature is thought to be related with the concept in search. If the first returned result does not coincide with the expectations, it would be passed for the analysis of the second result, and so forth, until is found one comment which provides effective information for establish a bridge between that Problem Domain concept and the Program Domain. The number of comments read for every query and its similarity to it was properly registered.

So, to initiate this test, it was necessary to load the source code of the program into **Darius**. Then it has was chosen an **IR** model to start with. For the sake of this test, every query was executed for the two different **IR** models, in order to see the differences between the results. These differences on the results and their proper discussion can be seen on Subsection 7.2.3.

With the program and **IR** model loaded, it was possible to look and search for the concepts, identified on Subsection 7.2.1. The first concept to search on the source code was the "PDF document" one, which identifies and represents the concept of a PDF file on *iText*. To do so it was executed the query *"PDF document"*. From the results returned by **Darius**, the comment which best fits the executed query was the one that can be seen on Figure 7.7. As it is observed on the comment, in *iText* a PDF document is represented by the class *PdfDocument* which is an instance of the class *Document*. From the comment it can also be concluded that a normal document is translated into a PDF document using an object of the class *PdfWriter*. So in order to see how to create a PDF document using a *PdfWriter* object it was executed the query *"PDF writer class"*. The best match to this query was the description of the *PdfWriter* class, which can be seen on Figure 7.8. Through the information on the comment it was concluded that this class is responsible for writing

a given document using the PDF language onto an output stream, which can be a file, or in this case a PDF file. So, in resume, to build a PDF file, it is necessary to create an object of the class *Document*, pass it onto an object of the class *PdfWriter* which will translate all of its elements onto the PDF language and into a file, which is physically a PDF file.

```
* <CODE>PdfDocument</CODE> is the class that is used by <CODE>PdfWriter</CODE>
* to translate a <CODE>Document</CODE> into a PDF with different pages.
* <P>
* A <CODE>PdfDocument</CODE> always listens to a <CODE>Document</CODE>
* and adds the Pdf representation of every <CODE>Element</CODE> that is
* added to the <CODE>Document</CODE>.
*
* @see                    com.itextpdf.text.Document
* @see                    com.itextpdf.text.DocListener
* @see                    PdfWriter
* @since      2.0.8 (class was package-private before)

public class PdfDocument extends Document {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/pdf/PdfDocument.java

3375
```

Figure 7.7: PDF Document query

```
* A <CODE>DocWriter</CODE> class for PDF.
* <P>
* When this <CODE>PdfWriter</CODE> is added
* to a certain <CODE>PdfDocument</CODE>, the PDF representation of every Element
* added to this Document will be written to the outputstream.</P>

public class PdfWriter extends DocWriter implements

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/pdf/PdfWriter.java

3783
```

Figure 7.8: PDF Writer class query

Then it was time to found out how to add information and contents to the document. First it was necessary to study the way metadata is added to the document, and how it is represented. To do so, this started by the execution of the query *"title document"*, to figure out how the title is added and represented within a document. The best match to this query was the comment represented on Figure 7.9. This is a comment to a method that adds a title, represented by a string, to an object of the *Document* class. Exploring this method within the file mentioned by **Darius**, the source code presented by this method is one that can be seen on Figure 7.10. Through this source code it was

concluded that meta information like the title, is added to the object *Document* by adding to a certain list an object of the class *Meta*, which constructor has the label of the type of metadata, in this case *Element.TITLE*, and the content of the metadata. To explore this class *Meta*, it was executed the query *"meta"*. On the list of results, one of the best matches was the comment of the header of the class *Meta*, observable on Figure 7.11. From this comment, it can be seen that the *Meta* class is a implementation of the class *Element*. It can also be seen that this class is also reserved for the subject, keywords, author and creation date, which were identified as part of the meta information concept of the *iText* Problem Domain. This last fact also concludes that it was not necessary to explore these other metadata types for **PC** purposes, as they are added to the *Document* object using the same paradigm, as observed on the source code of the *Document* class.

```
* Adds the title to a Document.
*
            * @param title
            *         the title
* @return    <CODE>true</CODE> if successful, <CODE>false</CODE> otherwise

public boolean addTitle(String title) {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/Document.java

12725
```

Figure 7.9: Document's title query

```java
public boolean addTitle(String title) {
    try {
        return add(new Meta(Element.TITLE, title));
            } catch (DocumentException de) {
        throw new ExceptionConverter(de);
    }
}
```

Figure 7.10: Add title to document method

Having figured out how to introduce meta information on a PDF file, is was time to explore the way *iText* manages the introduction of contents on the document. The concept of PDF content has been identified on Subsection 7.2.1 as a series of elements or high-level structures. The first element of content to identify was the anchor concept. To search for this concept on the code it was executed the query *"anchor element"*. The document or comment which best matched this

```
* This is an <CODE>Element</CODE> that contains
* some meta information about the document.
* <P>
* An object of type <CODE>Meta</CODE> can not be constructed by the user.
* User defined meta information should be placed in a <CODE>Header</CODE>-object.
* <CODE>Meta</CODE> is reserved for: Subject, Keywords, Author, Title, Producer
* and Creationdate information.
*
* @see                    Element
* @see                    Header

public class Meta implements Element {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/Meta.java

2238
```

Figure 7.11: Meta query

query was the one represented on Figure 7.12, which is the description for the class *Anchor*. In the comment it is described the way the user can build an anchor. It is also pointed out, and sustained by the source code attached, that the class *Anchor* is an extension of the class *Phrase*, which is also a concept of the Problem Domain of *iText*. To examine the nature of this concept on the source code it was executed the query *"phrase element"*. The best found match can be seen on Figure 7.13. This comment is the description for the header of the class *Phrase*, and provides several examples of how to build objects of this class. On the first sentence of the comment, it is also pointed out that a phrase is a series of chunks, which is also a concept from the Problem Domain. From the header of the class, it can be seen that the class *Phrase* is an extension of a list of objects from the class *Element*. So, crossing these two last facts, it can be predicted that a chunk is represented on the source code using a class, which is an extension or implementation of the *Element* class. To confirm this suspicion it was executed the query *"chunk element"*. The best result can be seen on Figure 7.14. This comment confirms the previous suspicion. This is the comment of the header of the *Chunk* class, which indicates that this class really implements the *Element class*. This comments also informs that a chunk is the smallest type of element that can be added to a given document.

Apart from these elements of content, there are others which were identified in the Problem Domain. One of them is the concept of paragraph. Once again to explore the implementation of this concept on the source code, it was executed a query to return comments which are more related. The executed query was the following: *"paragraph element"*. The comment with the best match to the query can be seen on Figure 7.15. In the comment and the attached source code it

```
* An <CODE>Anchor</CODE> can be a reference or a destination of a reference.
* <P>
* An <CODE>Anchor</CODE> is a special kind of <CODE>Phrase</CODE>.
* It is constructed in the same way.
* <P>
* Example:
* <BLOCKQUOTE><PRE>
* <STRONG>Anchor anchor = new Anchor("this is a link");</STRONG>
* <STRONG>anchor.setName("LINK");</STRONG>
* <STRONG>anchor.setReference("http://www.lowagie.com");</STRONG>
* </PRE></BLOCKQUOTE>
*
* @see                    Element
* @see                    Phrase

public class Anchor extends Phrase {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/Anchor.java
```

Figure 7.12: Anchor query

```
* A <CODE>Phrase</CODE> is a series of <CODE>Chunk</CODE>s.
* A <CODE>Phrase</CODE> has a main <CODE>Font</CODE>, but some chunks
* within the phrase can have a <CODE>Font</CODE> that differs from the * main <CODE>Font</CODE>.
* All the <CODE>Chunk</CODE>s in a <CODE>Phrase</CODE>
* have the same <CODE>leading</CODE>.
* <BLOCKQUOTE><PRE>
* // When no parameters are passed, the default leading = 16
* <STRONG>Phrase phrase0 = new Phrase();</STRONG>
* <STRONG>Phrase phrase1 = new Phrase("this is a phrase");</STRONG>
* // In this example the leading is passed as a parameter
* <STRONG>Phrase phrase2 = new Phrase(16, "this is a phrase with leading 16");</STRONG>
* // When a Font is passed (explicitly or embedded in a chunk), the default leading = 1.5 * size of the font
* <STRONG>Phrase phrase3 = new Phrase("this is a phrase with a red, normal font Courier, size 12", FontFactory.getFont(FontFacto
* <STRONG>Phrase phrase4 = new Phrase(new Chunk("this is a phrase"));</STRONG>
* <STRONG>Phrase phrase5 = new Phrase(18, new Chunk("this is a phrase", FontFactory.getFont(FontFactory.HELVETICA, 16, Font.B

public class Phrase extends ArrayList<Element> implements TextElementArray {
```

Figure 7.13: Phrase query

```
* This is the smallest significant part of text that can be added to a document.
* <P>
* Most elements can be divided in one or more <CODE>Chunk</CODE>s. A chunk
* is a <CODE>String</CODE> with a certain <CODE>Font</CODE>. All other
* layout parameters should be defined in the object to which this chunk of text
* is added.
* <P>
* Example:
* <STRONG>Chunk chunk = new Chunk("Hello world",
* FontFactory.getFont(FontFactory.COURIER, 20, Font.ITALIC, new BaseColor(255, 0,
* 0))); </STRONG> document.add(chunk);
*
* </PRE>
*
* </BLOCKQUOTE>

public class Chunk implements Element {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/Chunk.java
```

Figure 7.14: Chunk query

can be seen that a paragraph is represented by a class called *Paragraph*. The comment also points out that a phrase is a series of chunks. This is sustained by the attached source code, where it can be seen that the class *Paragraph* is an extension of the *Phrase*, which is also an extension for a List of objects of the class *Element*, whose example of instance is the *Chunk* class.

```
* A <CODE>Paragraph</CODE> is a series of <CODE>Chunk</CODE>s and/or <CODE>Phrases</CODE>.
* <P>
* A <CODE>Paragraph</CODE> has the same qualities of a <CODE>Phrase</CODE>, but also
* some additional layout-parameters:
* <UL>
* <LI>the indentation
* <LI>the alignment of the text
* </UL> *
* Example:
* <BLOCKQUOTE><PRE>
* <STRONG>Paragraph p = new Paragraph("This is a paragraph",
*           FontFactory.getFont(FontFactory.HELVETICA, 18, Font.BOLDITALIC, new Color(0, 0, 255)));</STRONG>
* </PRE></BLOCKQUOTE>
*
* @see                     Element
* @see                     Phrase
* @see                     ListItem

public class Paragraph extends Phrase {
```

Figure 7.15: Paragraph query

One of the other concepts of the Problem Domain to look for on the source code is the concept of section. Using the same paradigm of the the previous queries, the query used to search for this concept was the following: *"section element"*. The comment which seemed the best fit for this query can be seen on Figure 7.16. As it can be seen on the comment, this element is represented by the class *Section*. Through the comment and the header of the class, it can be seen that this class is a list of several other elements, which are other sections, paragraphs, lists or/and tables.

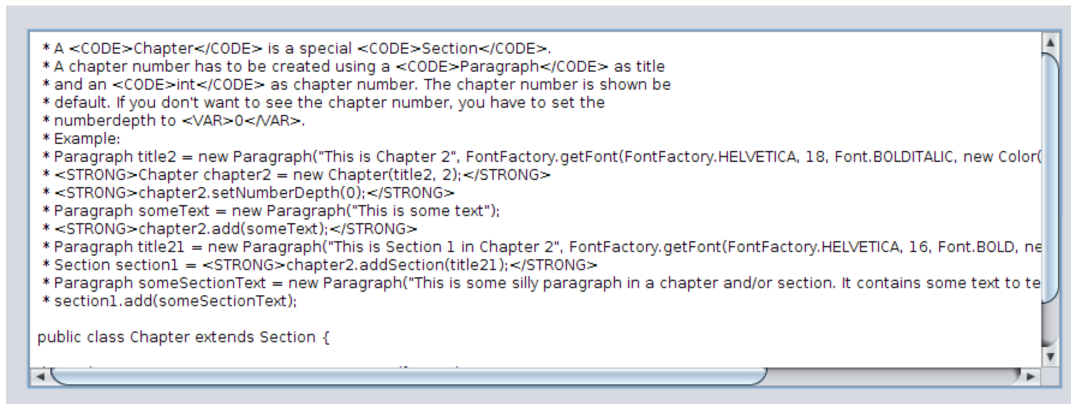
```
* A <CODE>Section</CODE> is a part of a <CODE>Document</CODE> containing
* other <CODE>Section</CODE>s, <CODE>Paragraph</CODE>s, <CODE>List</CODE>
* and/or <CODE>Table</CODE>s.
* Remark: you can not construct a <CODE>Section</CODE> yourself.
* You will have to ask an instance of <CODE>Section</CODE> to the
* <CODE>Chapter</CODE> or <CODE>Section</CODE> to which you want to
* add the new <CODE>Section</CODE>.
* Example:
* Paragraph title2 = new Paragraph("This is Chapter 2", FontFactory.getFont(FontFactory.HELVETICA, 18, Font.BOLDITALIC, new Color(
* Chapter chapter2 = new Chapter(title2, 2);
* Paragraph someText = new Paragraph("This is some text");
* chapter2.add(someText);
* Paragraph title21 = new Paragraph("This is Section 1 in Chapter 2", FontFactory.getFont(FontFactory.HELVETICA, 16, Font.BOLD, ne
* <STRONG>Section section1 = chapter2.addSection(title21);</STRONG>
* Paragraph someSectionText = new Paragraph("This is some silly paragraph in a chapter and/or section. It contains some text to te
* <STRONG>section1.add(someSectionText);</STRONG>

public class Section extends ArrayList<Element> implements TextElementArray, LargeElement {
```

Figure 7.16: Section query

Having figured out how the concept of section is materialized on the source code, it was time to

see how a similar concept is put on the source code: the chapter. The query executed was formed on the following way: *"chapter element"*. The best match to this query was the comment to the header of the class *Chapter*, which represents the chapter concept on the source code of *iText*. This comment can be seen on Figure 7.17. Though the information on the comment it could be concluded that a chapter is a special variety of a section. In the header of the class, this fact is sustained, because the class *Chapter* is indeed an extension of the *Section* class.

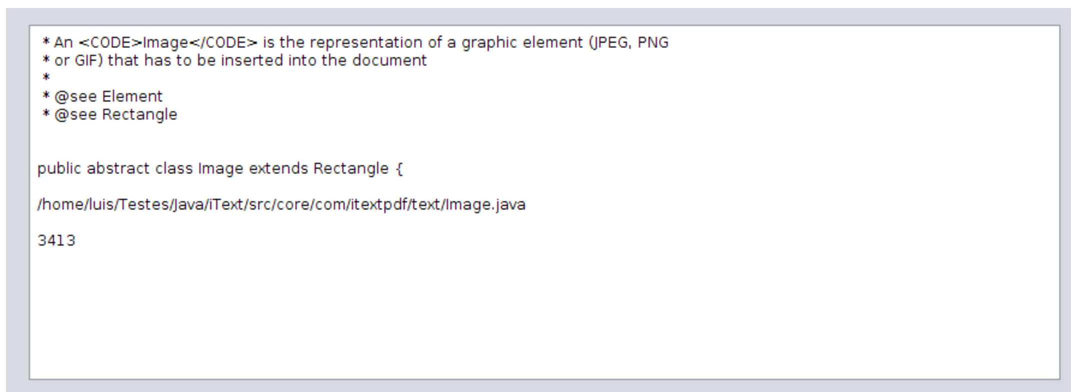


```
* A <CODE>Chapter</CODE> is a special <CODE>Section</CODE>.
* A chapter number has to be created using a <CODE>Paragraph</CODE> as title
* and an <CODE>int</CODE> as chapter number. The chapter number is shown be
* default. If you don't want to see the chapter number, you have to set the
* numberdepth to <VAR>0</VAR>.
* Example:
* Paragraph title2 = new Paragraph("This is Chapter 2", FontFactory.getFont(FontFactory.HELVETICA, 18, Font.BOLDITALIC, new Color(
* <STRONG>Chapter chapter2 = new Chapter(title2, 2);</STRONG>
* <STRONG>chapter2.setNumberDepth(0);</STRONG>
* Paragraph someText = new Paragraph("This is some text");
* <STRONG>chapter2.add(someText);</STRONG>
* Paragraph title21 = new Paragraph("This is Section 1 in Chapter 2", FontFactory.getFont(FontFactory.HELVETICA, 16, Font.BOLD, ne
* Section section1 = <STRONG>chapter2.addSection(title21);</STRONG>
* Paragraph someSectionText = new Paragraph("This is some silly paragraph in a chapter and/or section. It contains some text to te
* section1.add(someSectionText);

public class Chapter extends Section {
```

Figure 7.17: Chapter query

With this, only left the search for two concepts. One is the concept of image, that can be included into a PDF document. The query for the location of this concept was the following *"image element"*. The best result was the comment represented on Figure 7.18. An image is represented on the source code as a class called *Image* that is an extension of the class *Rectangle*, which is itself a implementation of the *Element* class.



```
* An <CODE>Image</CODE> is the representation of a graphic element (JPEG, PNG
* or GIF) that has to be inserted into the document
*
* @see Element
* @see Rectangle

public abstract class Image extends Rectangle {

/home/luis/Testes/Java/iText/src/core/com/itextpdf/text/Image.java

3413
```

Figure 7.18: Image query

The last concept of the Problem Domain to look for was the concept of list. To do so, the same

pattern of query was used: *"list element"*. The best match for this query was the comment of the header of the class *List*, which represents the list concept on the source code, and can can be seen on Figure 7.19. In this comment, it can be seen that an object of this class contains a list of objects of the class *ListItem*. To explore this class it was executed the query *"list item element"*. The best match to this query is represented on Figure 7.20. As it can be seen in the comment, the class *ListItem* is an extension of the class *Paragraph*, which as discussed before.

```
* A <CODE>List</CODE> contains several <CODE>ListItem</CODE>s.
* <B>Example 1:</B>
* <BLOCKQUOTE><PRE>
* <STRONG>List list = new List(true, 20);</STRONG>
* <STRONG>list.add(new ListItem("First line"));</STRONG>
* <STRONG>list.add(new ListItem("The second line is longer to see what happens once the end of the line is reached. Will it start on
* <STRONG>list.add(new ListItem("Third line"));</STRONG>
* </PRE></BLOCKQUOTE> *
* <B>Example 2:</B>
* <BLOCKQUOTE><PRE>
* <STRONG>List overview = new List(false, 10);</STRONG>
* <STRONG>overview.add(new ListItem("This is an item"));</STRONG>
* <STRONG>overview.add("This is another item");</STRONG>
*
* @see                    Element
* @see                    ListItem

public class List implements TextElementArray {
```

Figure 7.19: List query

```
* A <CODE>ListItem</CODE> is a <CODE>Paragraph</CODE> that can be added to a <CODE>List</CODE>.
* <B>Example 1:</B>
* <BLOCKQUOTE><PRE>
* List list = new List(true, 20);
* list.add(<STRONG>new ListItem("First line")</STRONG>);
* list.add(<STRONG>new ListItem("The second line is longer to see what happens once the end of the line is reached. Will it start on
* list.add(<STRONG>new ListItem("Third line")</STRONG>);
* <B>Example 2:</B>
* <BLOCKQUOTE><PRE>
* List overview = new List(false, 10);
* overview.add(<STRONG>new ListItem("This is an item")</STRONG>);
* overview.add("This is another item");
*
* @see        Element
* @see        List
* @see        Paragraph

public class ListItem extends Paragraph {
```

Figure 7.20: List Item query

Having finished all the search for the Problem Domain concepts on the source code with success, it was possible to aggregate all of the information that was able to be extracted by **Darius**. On Figure 7.21 it can be seen all of the information extracted, organized in a Class diagram, using Unified Modeling Language (UML). All of the classes and interfaces identified on the diagram, represent concepts of the Problem Domain, which were bridged into the Program Domain. These

bridges are represented by the multiple connections and stereotypes, that can be visualized on the diagram.

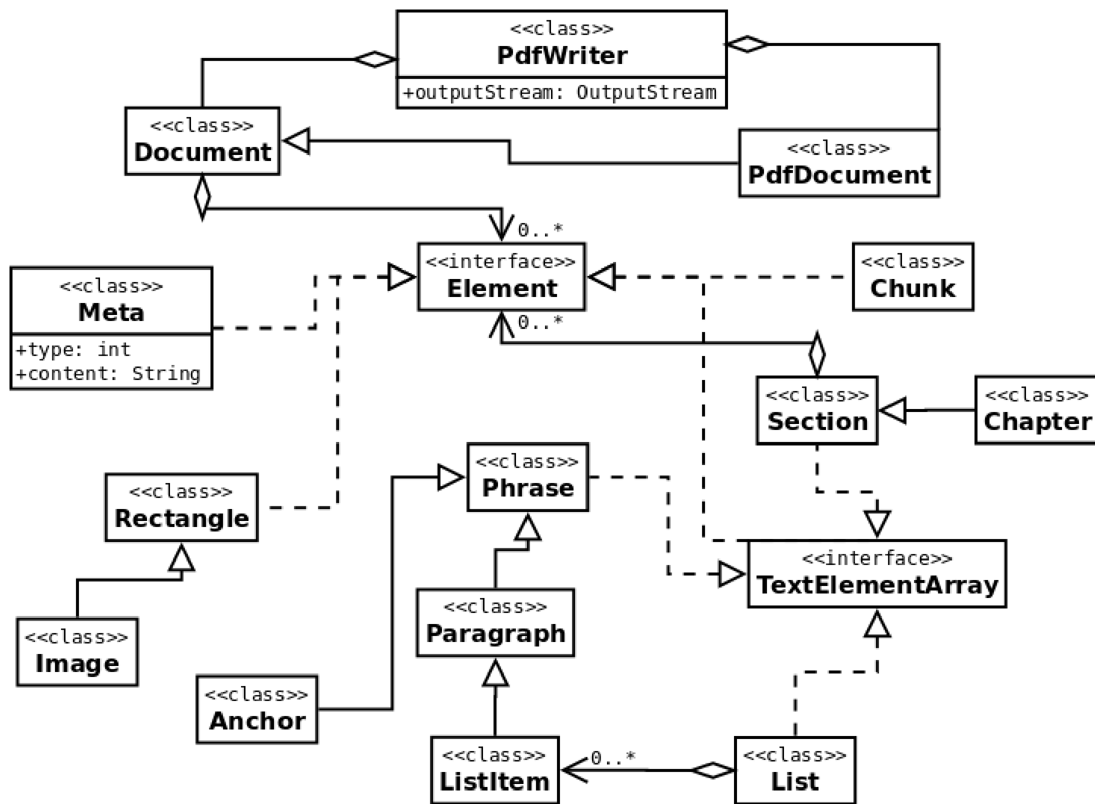


Figure 7.21: UML Class Diagram of the results

7.2.3 Discussion of the results

On the last subsection it was executed a test or case study of a program called *iText* using **Darius**. The test started with the identification of the concepts of the Problem Domain of the program, which can be seen on Figure 7.6. Step by step, and using the concept locator of **Darius**, it was able to figure out the implementation of every concept on the source code and the relationships among them, using the information presented on comments. In the end, it was able to aggregate all of the collected information, and create a UML class diagram, which can be seen on Figure 7.21, that gives a good overview of the bridges between the Problem and the Program Domain, or in other words, the implementation of a certain Problem Domain concept on the source code.

The conclusion that can be extracted from this test, is that **Darius** provides a good user experience and more efficiency to the process of comprehending a given program. Using only simple but

	VSM		LSA	
Query	**Sim.**	**Order**	**Sim.**	**Order**
"PDF document"	0.351	22	0.04	83
"PDF writer class"	0.321	35	0.06	70
"title document"	0.489	7	0.011	17
"meta"	0.373	9	0.001	30
"anchor element"	0.478	2	0.003	123
"phrase element"	0.428	20	0.009	24
"chunk element"	0.329	39	0.004	164
"paragraph element"	0.323	23	0.006	57
"section element"	0.302	40	0.006	61
"chapter element"	0.281	58	0.002	153
"image element"	0.38	24	0.006	25
"list element"	0.292	89	0.005	94
"list item element"	0.426	15	0.008	32
Average	0.367	29	0,012	72

Table 7.1: Results from the Queries

effective queries, the process of locating concepts using comment information was faster that the complex task of reading the whole source code of the program. To sustain this fact are presented on Table 7.1 the similarity and order of the best match comment to each of the executed queries, on the two types of **IR** models that **Darius** provides.

As it can be seen on Table 7.1, using the **VSM** model, there was an average of 29 comments that had to be read in order to locate the perfect location of a given Problem Domain concept. Considering that *iText* has a total of 13343 comments, this means that in average only 0.21% of the comments had to read. However, the results declined in quality passing for the **LSA** model, where there was an average of 72 read comments, in order to find the best match. Although **LSA** scored worse results, the percentage of the average read comments is still minimal, obtaining just 0.54% of all the comments presented on the source code of *iText*.

Analysing the results of the similarities on Table 7.1, it can also be concluded that the values of the percentage of comments read could be different if the queries were perfected and upgraded. In the **VSM** model, the executed queries scored only an average of 36.7%, almost one third of the full potential. Strangely, the results of similarity for the **LSA** model, scored a minimal value of 1.2%. The vast difference to the result of the **VSM** model, gives an opportunity for further investigation, for the search of answers for this question.

Combining the results from the similarity and order of best match for the two types of **IR** models,

there is no doubt, that in this test, the **VSM** model was more effective and retrieved faster results than the **LSA** model. Given the true potential of the **LSA** model, further investigations must be made, using different types of queries, or testing different numbers of dimensions for the concept space, in order to see whether this model can present better results.

In resume, the results from Table 7.1, sustained and reinforced the true potential of **Darius**, as a concept locator using comment information, and as a **PC** tool. Considering the tasks at hand, this tool provided minimal effort to the user on the development of concept location processes, reducing the work to do to just 0.21% of the total of comment information, needed for the process of comprehending the program.

7.3 Summary

Having proved as feasible to develop, the second version of **Darius** introduced **IR** techniques that addressed the *concept assignment problem*.

The process used to developed this second version was very similar to other approaches that addressed the same problem. It started with the representation of the *logical view of the documents* or the document database. To do it, it was created a matrix that associated the weight of a certain word in a document, which in this case, was a comment (granularity level).

Having defined the document database, it was introduced two models of **IR**, in order to see what type would deliver the best results. In particular it was implemented the **VSM** and the **LSA**. There was also implemented methods of calculating similarity between queries and comments, in order to establish the ranks of comments. It was also created a graphical interface that included an area where the user can introduce the desired query, and see the comments more related to that query, according to the **IR** model loaded.

In order to test **Darius** as a **PC** tool, it was used a program called *iText*, which is a Java library that manipulates and creates PDF documents. Having previously defined the Problem Domain of the program, and the singular concepts behind it, the idea was, through appropriate queries, search for the materialization of these concepts on the source code, using the information included on comments. In the end, there could be established all of the bridges between the Problem and the Program Domain.

The result have shown that the test was a success. All of the Problem Domain concepts, pre-

viously defined, were able to be found on the source code, and with the information extracted it was able to create a diagram that clearly shows the relation between the Problem and the Program Domain of *iText*. On terms of results from the similarities and from the order of the best matches, the **VSM** scored better results. In particular, with this model, in average, only 0.21% of the comments had to be read in order to find the appropriate information about a certain problem domain concept. The results have proven that **Darius** was able to catalyze the process of comprehending the program, and also showed that the information extracted from the comments, have provided strong means of obtaining that comprehension.

Part V

Conclusion

Chapter 8

Conclusion

This document is a dissertation, from a master degree in Informatic Engineering, included in the areas of Program Comprehension and Comment Analysis. Throughout the first, second and third parts of this dissertation it was given a detailed but concise description of the state-of-the-art of these two areas. The fourth part had the objective of providing the answers for the questions raised and the results for the goals aimed by this work.

In this chapter, there are given some final thoughts and conclusions about the work done in this project, the discussion of the results, and also are given topics of interest of further future investigations. Below, there is described and summarized each of the sections presented in this document.

Chapter II - Program Comprehension In this chapter it is described an exhaustive study from the **PC** area. Every main topic of this area was addressed, starting with the definition of mental model, and the enumeration and full description of its static and dynamic elements, and also the cognitive enablers. It was also given an overview of knowledge domain, focusing on the Problem and on the Program Domain topics. In this chapter it was also given much attention to the the area of Cognitive models, describing in detail the more important ones and that are more referenced on the literature. The chapter ends with an enumeration of a list of **PC** tools, that use program comprehension concepts to aid programmers understand programs.

Chapter III - Information Retrieval for Program Comprehension In this chapter it is made a bridge between the areas of **PC** and **IR**. The chapter starts with an overview of the **IR** area,

discussing the main process behind it and including a detailed description of the main models used on **IR**. The chapter ends with an exhaustive study about the introduction of **IR** techniques on **PC** approaches and tools, that are available on the literature.

Chapter IV - Source Code Comments Throughout this chapter it is given the look of the source code comment on the literature. This exhaustive study includes the enumeration of the different kinds of comments, the discussion of the different types of comment content, and it is also given an overview of the literature of the relation between the source code comment and the natural language, the program size, and its density and practice on the Informatics field.

Chapter V - Comment Analysis for Program Comprehension In this chapter it is presented an overview of the analysis of comments on the **PC** area. This chapter starts with an exhaustive study, extracted from the literature, of the role of comments on the process of comprehension of a given program. The chapter ends with an enumeration of the available approaches that use comment analysis to enhance **PC**.

Chapter VI - Darius: The first stage In this chapter it is introduced **Darius**, the tool developed in this master degree project. In particular, this chapter gives a full description of a preliminary study that tried to prove the feasibility of developing that tool. There are included the goals of this study and the strategies to execute in order to achieve them. It includes the description of **Darius** first version that executed those tests, detailing each one of its components and the graphical interface. The description of the steps and results of this preliminary study conclude this chapter.

Chapter VII - Darius: The second stage In this chapter it is included the exhaustive description of the second version **Darius**, as a concept locator. Each one of the components and the graphical interface are fully detailed within this chapter. To prove the efficiency of this tool, it was executed a test or case study that is described in the end of this chapter. This description includes all the steps given and the discussion of the obtained results.

8.1 Discussion and Conclusions

The main goal of the work developed and described in this master dissertation was to check if the information regarding the Problem and the Program Domain included on source code comments, could catalyzed the process of understanding a given program. To do so, it was developed a tool called **Darius** that used **IR** techniques and applied them to the contents of source code comments, in order to provide means for a faster and more effective comprehension of programs.

From the results obtained through a test using **Darius**, at a first glance, the idea is that the tool provided effective means to obtain a faster and more effective comprehension of a program, comparing it to the simple act of reading the whole source code of the program. Considering that the program at test included a total of 13343 comments in a total of 480 files of source code, the results from **Darius**, concluded that, in the minimum, there was only an average of 0.21% of those comments that had to be read in order to obtain the comprehension of a single Problem Domain concept. In addition, the results obtained of the similarity of the executed queries, which where rather low, lead to say that this percentage of comments read, could be even lower, if the queries were perfected. So, the full potential of this tool as a **PC** catalyzer was not put totally to the test, which could be done in a future work. However, this tool included the loading of a **LSA** model, which is referenced in the literature as the most effective of the **IR** models, that returns the best results. Although the results obtained in this model are not to be discarded, there is a slight disappointment on the **LSA** performance, which leads to further opportunities of investigations, with the goal of checking the reasons of such results.

In resume, the results obtained from the test applied to **Darius**, give reasons for affirming positively to Question 4 from Section 1.2 that asked if *the information regarding the problem and program domain contained on comments, properly retrieved and analyzed by a Comment Analysis PC tool, can be used to enhance the program comprehension by establishing a bridge between the two domains*. The reality from the results from the test applied to **Darius**, is that from simple queries it was possible to find comments which best fitted that query, and from that comments it was possible to extract information that provided means of establishing bridges between the Problem and Program domain, or in other words, the materialization of a certain Problem Domain concept on the source code of the program.

Although this test or case study that was applied on **Darius** provided means of proving its

effectiveness as a **PC** tool, the reality and common sense demands that it must not be generalized the idea that **Darius** works for every case, and the intention of this dissertation is to keep that clear. In particular, **Darius** needs harder and more complex tests with other types of programs and software projects, in order to see its full potential, and to check eventual flaws, that could be solved. Looking at **Darius** as a **PC** tool, it is also necessary to put it at test using human subjects, in order to check if, in general, the users and programmers would benefit in using this tool for program understandability purposes. Only with the results provided by this final test, it could assumed and sustained completely the idea that **Darius** is able to provide effective means of obtaining **PC**.

The steps taken to obtain comprehension of the program at test, included several strategies that tend to followed a top-down model of comprehension, particularly the *Brooks* model. However, this does not indicate that **Darius** is only targeted for those types of approaches, in relegation for others. Again, it is reinforced the idea that **Darius** was not fully put to the test, included the use of other approaches and strategies of comprehension, to fulfill the goals.

Before the development of the second version of **Darius**, which included this **PC** module, it was developed a first version that included means of proving the feasibility of the development of that second version. With that first version it was performed a preliminary study that explored the quantity and contents of the comments on a set of programs and software projects. The results proved that there was at least one program that responded to the requirements, and so it was able to be explored for comprehensibility purposes, and the second version of **Darius** was proved as feasible to develop. It is important to point out that the values that were used as references to the two different tests, are not standards or fixed values accepted in the literature. These values are just virtual points of reference that were extracted from works in the same field, that are available on the literature, and that were properly referenced in this dissertation. The idea of this preliminary study, was, as its name says, to preliminary study the nature of the raw material deal with, which were source code comments. So, the decision was that it would be extremely abrupt to proceed right way for the development of a **PC** tool, without executing this preliminary study. In addition, the incorporation of the statistical functions about the comment part of the source code, strengthened the power of this tool as a Comment Analysis one. Apart from just being a **PC** tool, with these powers, **Darius** can also be used for simple analysis of the comments on the source code of a program, which can be useful, for example, to check software quality, if the commenting is a requirement of it.

Concluding this master disseration, the idea that stands is that the information from comments can be explored, and with that, the programmer can use it for program comprehension purposes, by establishing bridges between the Problem and the Program Domain. The objective of this work was to explore this power, and reinforce it, by creating a tool that catalyzes this process. **Darius** proved to be an effective tool, and with that effectiveness, all the goals aimed by this work were fulfill with success.

8.2 Relevant work

At the same time the work decribed in this dissertation was developed, there was also executed other activities that are related and relevant to this work. In particular, this included the publication of an article on a conference that can be seen in [37].

8.3 Future work

In the end of Section 8.1 it is stated that all the objectives and goals, previously defined in the beginning of this work, were fulfilled. However, throughout the study and development there was some points of potential improvement or further investigations, properly referenced. Below there is a list of potential topics of further investigation and improvement, and for future works:

- Investigation and improvement of the **LSA** model developed on **Darius**. The results from the test conducted with the **LSA** model have disappointed, taking in consideration that this **IR** model is referenced as one of the most effectives. In particular, there has to be tested with different numbers of dimensions for the concept space, in order to check what value is the optimal;
- Development and incorporation of other types of **IR** models, in order to check the effectiveness of more complex models;
- Execution of tests with different and upgraded queries. The executed test included queries which in average scored an average of 36.7% on terms of similarity to the best match. The improvement of these queries could return better results for similarities, reducing the number of comments to be read, improving the pace of the comprehension process.

- Execution of a study with human subjects of the **Darius** tool, in order to see the help that this tool would provide for real programmers, on their tasks of comprehending real programs. The reaction to this tool, would show the true potential of **Darius**.

- Improvement of the graphical interface to provide better user experience to programmers, and improvement of the back-end modules, reducing the time of computation and the use of resources, such as memory.

Bibliography

[1] Inference-based and expectation-based processing in program comprehension. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 71–, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Program comprehension by visualization in contexts. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 332–, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Hervé Abdi. *Singular Value Decomposition (SVD) and Generalized Singular Value Decomposition (GSVD)*. Sage, 2007.

[4] Hirohisa Aman and Hirokazu Okazaki. Impact of comment statement on code stability in open source development. In *Proceeding of the 2008 conference on Knowledge-Based Software Engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering*, pages 415–419, 2008.

[5] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the canto environment. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 72–81, oct 1997.

[6] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 2000.

[7] Oliver Arafat and Dirk Riehle. The commenting practice of open source. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 857–864, 2009.

[8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] Victor R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Softw.*, 7:19–25, January 1990.

[10] Nicolas J. Belkin. Helping people find what they don't know. *Commun. ACM*, 43, August 2000.

[11] Mario Berón. Program inspection to interconnect the behavioral and operational views for program comprehension. Technical report, National University of San Luis & University of Minho, 2009.

[12] Mario Berón, Pedro R. Henriques, Maria J. V. Pereira, Roberto Uzal, and G. Montejano. A Language Processing Tool for Program Comprehension. In *CACIC'06 - XII Argentine Congress on Computer Science, Universidad Nacional de San Luis, Argentina*, 2006.

[13] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. Technical report, 1994.

[14] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, 1993.

[15] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[16] George E. P. Box and George C. Tiao. *Bayesian Inference in Statistical Analysis (Wiley Classics Library)*. Wiley-Interscience, April 1992.

[17] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, 1978.

[18] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[19] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Václav Rajlich. Jripples: A tool for program comprehension during incremental change. In *in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05*, pages 149–152, 2005.

[20] I. Burnstein, N. Jani, S. Mannina, J. Tamsevicius, M. Goldshteyn, and L. Lendi. The development of a knowledge-based software fault localization system. In *Systems, Man and Cybernetics, 1992., IEEE International Conference on*, pages 317–322 vol.1, oct 1992.

[21] G. Canfora, L. Mancini, and M. Tortorella. A workbench for program comprehension during software maintenance. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 30–39, mar 1996.

[22] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins. What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14, January 1999.

[23] Ned Chapin, Joanne E. Hale, Khaled M. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, January 2001.

[24] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[25] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[26] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28:294–306, June 1989.

[27] Andrew M. Dearden and Derek G. Bridge. Choosing a reasoning style for knowledge based system: lessons from supporting a help desk. *Knowledge Engineering Review*, 8, 1993.

[28] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[29] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1): 1–38, 1977.

[30] Susan T. Dumais. Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1):188–230, 2004.

[31] Katrin Erk and Sebastian Padó. A structured vector space model for word meaning in context. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 897–906, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[32] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[33] Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. An approach to program understanding by natural language understanding. *Nat. Lang. Eng.*, 5:219–236, September 1999.

[34] Letha H. Etzkorn, Carl G. Davis, and Lisa L. Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731–1756, 2001.

[35] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In *Proceedings GUIDE 48*, April 1983.

[36] Beat Fluri, Michael Würsch, and Harald Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.

[37] José L. Freitas, Pedro R. Henriques, and Daniela da Cruz. The role of comments on program comprehension. In Luis Caires and Raul Barbosa, editors, *INForum'11 — Simpósio de Informática (CoRTA'11 track)*, September 2011.

[38] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. 2005.

[39] Sonia Haiduc and Andrian Marcus. On the use of domain terms in source code. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 113–122, 2008.

[40] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, pages 50–57, 1999.

[41] Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 179–180, 2006.

[42] Dean Jones, Trevor Bench-capon, and Pepijn Visser. Methodologies for ontology development. pages 62–75, 1998.

[43] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. Computing Mcgraw-Hill, January 1978.

[44] April Kontostathis and William M. Pottenger. A framework for understanding latent semantic indexing (lsi) performance. *Inf. Process. Manage.*, 42:56–73, January 2006.

[45] Rainer Koschke, Andrian Marcus, and Gerald Gannod. Guest editor's introduction to the special section on the 2009 international conference on program comprehension (icpc 2009). *Software Quality Journal*, 19, 2011.

[46] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, 1999.

[47] Thomas K. Landauer. *Latent Semantic Analysis*. John Wiley & Sons, Ltd, 2006.

[48] Thomas K. Landauer, P. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(1):259–284, 1998.

[49] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 672–673, 2005.

[50] Dik L. Lee, Huei Chuang, and Kent Seamons. Document ranking and the vector-space model. *IEEE Softw.*, 14:67–75, March 1997. ISSN 0740-7459.

[51] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 350–357, 1976.

[52] Stanley Letovsky. Cognitive processes in program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[53] P.K. Linos and V. Courtois. A tool for understanding object-oriented program dependencies. In *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, pages 20–27, nov 1994.

[54] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[55] W. Lowe and T. Panas. Rapid Construction of Software Comprehension Tools. In *International Journal of Software Engineering and Knowledge Engineering*, 2005.

[56] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, 2008.

[57] Rasmus E. Madsen, David Kauchak, and Charles Elkan. Modeling word burstiness using the dirichlet distribution. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 545–552, New York, NY, USA, 2005. ACM.

[58] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, 2000.

[59] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 103–112, 2001.

[60] Jonathan I. Maletic and Naveen Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE international conference on Automated software engineering*, 1999.

[61] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[62] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[63] Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004.

[64] Andrian Marcus, Václav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.

[65] Jeff Michaud, Margaret-Anne D. Storey, and Hausi Müller. Integrating information sources for visualizing java programs. In *In Proc. of the International Conference on Software Maintenance (ICSM'01)*, page 250. IEEE, 2001.

[66] Bruce Momjian. *PostgreSQL: introduction and concepts*. 2001.

[67] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13 (6):47–60, November 1996.

[68] Hausi Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th international conference on Software engineering*, ICSE '88, pages 80–86, 1988.

[69] Natalya F. Noy and Deborah L. mcguinness. Online, 2001.

[70] E. Nurvitadhi, E. Nurvitadhi, and C. Cook. Do class comments aid java program understanding? *Frontiers in Education, Annual*, 1:T3C13–17, 2003.

[71] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 331–341, 2009.

[72] Nancy Pennington. Empirical studies of programmers: second workshop. chapter Comprehension strategies in programming, pages 100–113. 1987.

[73] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[74] Maria J. V. Pereira, Marjan Mernik, Daniela da Cruz, and Pedro R. Henriques. Program comprehension for domain-specific languages. *Comput. Sci. Inf. Syst.*, 5(2):1–17, 2008.

[75] Ari Pirkola. The effects of query structure and dictionary setups in dictionary-based cross-language information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 55–63, 1998.

[76] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, 2007.

[77] Ruben Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[78] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–, 2002.

[79] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a tool suite for program analysis and reverse engineering. In Luís Pinho and Michael González Harbour, editors, *Reliable Software Technologies – Ada-Europe 2006*, volume 4006 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin / Heidelberg, 2006.

[80] Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, 60:2004, 2004.

[81] Spencer Rugaber. Program Comprehension, May 1995.

[82] Spencer Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9:143–192, January 2000.

[83] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18:613–620, November 1975.

[84] Carolyn B. Seaman. Software maintenance: Concepts and practice authored by penny grubb and armstrong a. takang world scientific, new jersey. 2003; 349 pages isbn 981-238-426-x (paperback) us$40. *J. Softw. Maint. Evol.*, 20:463–466, November 2008.

[85] Teresa M. Shaft and Iris Vessey. The relevance of application domain knowledge: characterizing the computer program comprehension process. *J. Manage. Inf. Syst.*, 15:51–78, June 1998.

[86] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8: 219–238, 1979.

[87] Elliot Soloway and Kate Ehrlich. *Empirical studies of programming knowledge*, pages 235–267. ACM, New York, NY, USA, 1989.

[88] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and Processes in the Comprehension of Computer Programs. *The Nature of Expertise*, pages 129–152, 1988.

[89] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 653–662, 2008.

[90] D. J. Spiegelhalter, A. Thomas, N. G. Best, W. R. Gilks, and D. Lunn. Bugs: Bayesian inference using gibbs sampling. 2003.

[91] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12: 43–60, 2002.

[92] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44:171–185, January 1999.

[93] Margaret-Anne D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.

[94] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. *SIGOPS Oper. Syst. Rev.*, 41, October 2007.

[95] Ted Tenny. Procedures and comments vs. the banker's algorithm. *SIGCSE Bull.*, 17:44–53, September 1985.

[96] Ted Tenny. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, September 1988.

[97] Michael L. Van De Vanter. The documentary structure of source code, 2002.

[98] Bradley L. Vinz and Letha H. Etzkorn. Improving program comprehension by combining code understanding with comment understanding. *Know.-Based Syst.*, 21:813–825, December 2008.

[99] Anneliese von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, 1994.

[100] Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. 28(8):44–55, 1995.

[101] Andrew Walenstein. Cognitive support in software engineering tools: A distributed cognition framework, 2002.

[102] Alf Inge Wang and Erik Arisholm. The effect of task order on the maintainability of object-oriented software. *Inf. Softw. Technol.*, 51:293–305, February 2009.

[103] Susan Wiedenbeck and Nancy J. Evans. Beacons in program comprehension. *SIGCHI Bull.*, 18:56–57, October 1986.

[104] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223, 1981.

[105] Peter Young. Program Comprehension, May 1996.