Thesis
Doctoral Program in Information Science and Technology
Artificial Intelligence

Context-Based Retrieval in Software Development

Bruno Emanuel Machado Antunes

Thesis Advisor
Prof. Paulo Jorge de Sousa Gomes

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

2012

This thesis was supported by a FCT (Fundação para a Ciência e Tecnologia)
scholarship grant, with reference SFRH/BD/43336/2008,
co-funded by ESF (European Social Fund).

To my parents, Ana and José, and my love, Vera.

Abstract

Although software development may include all the activities that result in a software product, from its conception to its realization, here we focus on the process of writing and maintaining the source code. Software development projects have been increasing in size and complexity, requiring developers to cope with a large amount of contextual information during their work. With workspaces frequently comprising hundreds, or even thousands, of artifacts, they spend a considerable amount of time navigating the source code or searching for a specific source code artifact they need to work with. With the aim of helping developers understand the source code structure and find what they need, modern Integrated Development Environments (IDEs) provide several features for searching and navigating the source code. But, according to some studies, developers still spend a considerable amount of time searching and navigating the source code structure.

With regard to search, the most commonly used approach is the matching of specific patterns in the lines of code that comprise a software system, requiring a direct correspondence between the pattern and the text in the source code. The limitations of this approach have been surpassed by the research carried out in the field of Information Retrieval (IR), encouraging researchers to use these techniques to help developers finding relevant source code for their current task. But, despite the fact that context is argued to improve the effectiveness of IR systems, as far as we know, none of the previous approaches have used the contextual information of the developer to improve the retrieval or ranking of relevant source code in the IDE.

Another interesting form of delivering relevant source code artifacts to a developer is using a recommender system. These systems have been used in a wide variety of domains, to help users find relevant information, deal with information overload and provide personalized recommendations of very different kinds of items. In software development, researchers studied ways of using contextual information to recommend source code artifacts that are potentially relevant for the current task of the developer. But, these approaches usually use a limited context model, or require contextual information to be explicitly provided.

The research described in this thesis is focused on the development of a context-based approach to search and recommendation of source code in the IDE. The source code structure stored in the workspace of the developer is represented in a knowledge base. A context model represents the source code elements that are more relevant for the developer in a specific moment. These structures are then used to improve the retrieval and ranking of source code elements, such as classes, interfaces and methods, taking into account their relevance to the current context of the developer. The relevance of the source code elements retrieved is computed based on the structural and lexical relations that exist between these elements and the elements in the context model.

We have implemented a prototype that implements and integrates our approach in the Eclipse IDE. This prototype was tested with a group of developers in order to validate our approach. The statistical information collected shows that the source code elements manipulated by the developer are highly related. This supports our claim that the relations that exist between source code artifacts can be used to measure the proximity between these artifacts, and to compute their relevance in the current context of the developer. Also, we have verified that the context components have a clear contribution to improve the ranking of search results, with the search results selected by the developers using our approach being better ranked in more than half of the times. With respect to recommendations, although the results are not so evident, we have shown that our context model can be used to retrieve relevant source code elements for the developer, being able to predict the needed source code element in more than half of the times.

Keywords: Context, Software Development, Ontologies, Information Retrieval, Recommender Systems.

Resumo

Apesar do desenvolvimento de software poder incluir todas as actividades que dão origem a um produto de software, desde a sua concepção até à sua realização, neste trabalho focamo-nos apenas no processo de desenvolvimento e manutenção do código fonte. Os projectos de desenvolvimento de software têm crescido em tamanho e complexidade, exigindo que os programadores lidem com uma elevada quantidade de informação contextual durante o seu trabalho. Com ambientes de trabalho frequentemente constituídos por centenas, ou mesmo milhares, de artefactos, os programadores gastam uma quantidade de tempo considerável a navegar no código fonte ou a procurar um artefacto específico com o qual precisam trabalhar. Com o objectivo de ajudar os programadores a compreender a estrutura do código fonte e a encontrar os artefactos que precisam, os ambientes de desenvolvimento integrados (IDEs) actuais fornecem diversas ferramentas para procurar e navegar no código fonte. No entanto, de acordo com alguns estudos, os programadores gastam ainda assim uma quantidade de tempo considerável a pesquisar e navegar na estrutura do código fonte.

Relativamente à pesquisa, a abordagem utilizada mais frequentemente é a procura de padrões nas linhas de código de um sistema de software, o que requer uma correspondência directa entre o padrão pesquisado e o texto no código. As limitações desta abordagem foram já ultrapassadas por técnicas desenvolvidas na área de recolha de informação, encorajando vários investigadores a usar estas técnicas para ajudar os programadores a encontrar artefactos relevantes para as suas tarefas. No entanto, apesar de ter sido defendido que o contexto aumenta a eficácia dos sistemas de recolha de informação, tanto quanto sabemos, nenhuma das abordagens anteriores utilizou o contexto do programador para melhorar a recolha ou a ordenação de resultados de pesquisa de código fonte no IDE.

Outra forma interessante de ajudar os programadores a encontrar artefactos de código fonte relevantes para as suas tarefas, é utilizando sistemas de recomendação. Estes sistemas foram já utilizados numa grande variedade de domínios, para ajudar os utilizadores a encontrar informação relevante, lidar com a sobrecarga de informação e providenciar recomendações de tipos muito diferentes de items. Em desenvolvimento de software, vários investigadores estudaram formas de utilizar a informação contextual do programador para recomendar artefactos de código fonte potencialmente relevantes para as suas tarefas. No entanto, estas abordagens utilizam normalmente um modelo de contexto limitado, ou requerem que o programador forneça de forma explícita a informação contextual.

A investigação descrita nesta tese centra-se no desenvolvimento de uma abordagem baseada no contexto para pesquisa e recomendação de código fonte num IDE. A estrutura do código fonte armazenado no ambiente de trabalho do programador é representada numa base de conhecimento. Um modelo de contexto representa os elementos de código fonte que são mais relevantes para o programador num determinado momento. Estas estruturas são posteriormente utilizadas para melhorar a recuperação e classificação de elementos de código fonte, tais como classes, interfaces e métodos, tendo em conta a sua relevância para o contexto actual do programador. A relevância dos elementos de código fonte recolhidos é calculada usando as relações estruturais e lexicais, que existem entre estes elementos e os elementos no modelo de contexto do programador.

Foi também implementado um protótipo que integra a nossa abordagem no Eclipse IDE. Este protótipo foi depois testado com um grupo de programadores, de forma a validar a nossa abordagem em ambiente real. A informação estatística recolhida demonstra que os elementos de código fonte manipulados pelo programador estão altamente relacionados. O que suporta a nossa ideia de que as relações que existem entre os artefactos de código fonte podem ser utilizadas para avaliar a proximidade entre estes artefactos, e calcular assim a sua relevância no contexto actual do programador. Também verificámos que as componentes do contexto têm uma clara contribuição para melhorar a classificação dos resultados da pesquisa, sendo que os resultados seleccionados pelos programadores, utilizando a nossa abordagem, estavam melhor classificados em mais de metade das vezes. No que diz respeito às recomendações, apesar dos resultados não serem tão evidentes, demonstrámos que o nosso modelo de contexto pode ser utilizado para identificar elementos de código fonte relevantes para o programador, sendo capaz de prever o próximo elemento acedido pelo programador em mais de metade das vezes.

Palavras-chave: Contexto, Desenvolvimento de Software, Ontologias, Recolha de Informação, Sistemas de Recomendação.

Acknowledgements

Although this thesis represents the culmination of a personal journey, it would not be possible without the support of some people, who have helped me, one way or another, to achieve this goal.

First of all, I would like to thank my advisor, Professor Paulo Gomes, for taking me as his student and trusting my work. He was always there when I needed advice and guidance, helping me find the solution when I only saw the problem. More than a mentor, he became a friend and a source of inspiration.

This work would not be possible without the contribution of the research fellows Joel Cordeiro, Francisco Correia e Pedro Costa. I would like to thank them for helping to shape some of the ideas that were explored in this work. I would also like to thank all the volunteers who have accepted to use our prototype during their work. Their contribution was essential to validate and improve our approach.

At a more personal level, I would like to thank Vera, for having supported me in all the bad moments, for sharing with me the happiness of the good times, and for understanding all the moments I had to be absent. I would not be where I am today without the support of my parents, Ana and José, whom I would like to thank for teaching me to set goals, for encouraging me to fight for them, and for giving me everything I ever needed to achieve them. They are, and will always be, my reference of courage and determination. Last, but not least, I leave a word of gratitude to my family and friends, for their continuous and unconditional support.

This thesis was supported by a FCT (Fundação para a Ciência e Tecnologia) scholarship grant, with reference SFRH/BD/43336/2008, co-funded by ESF (European Social Fund). I would like to thank FCT for providing me the financial conditions I needed to do what I like the most, research. Also, I thank CISUC (Centre for Informatics and Systems of the University of Coimbra), for hosting me and giving me the best conditions to develop and promote my work.

Finally, I would like to thank all those who, although not mentioned here, have crossed with me during the long walk that brought me here.

Contents

Chapter 1: Introduction **1**
1.1 Research Goals 3
1.2 Approach 4
1.3 Contributions 5
1.4 Outline 6

Chapter 2: Background Knowledge **7**
2.1 Ontologies 7
2.1.1 Classification 8
2.1.2 Building Process 9
2.1.3 Applications 10
2.2 Context 11
2.2.1 What is context? 11
2.2.2 Context Modeling 12
2.3 Information Retrieval 14
2.3.1 Modeling 15
2.3.2 Text Processing 17
2.3.3 Query Formulation 18
2.3.4 Evaluation 19
2.3.5 Context-Aware Information Retrieval 20
2.4 Recommender Systems 21
2.4.1 Classification 21
2.4.2 Evaluation 22
2.4.3 Context-Aware Recommendation 23
2.5 Software Development 24
2.5.1 Search 28
2.5.2 Recommendation 29
2.6 Summary 31

Chapter 3: Approach **33**
3.1 Knowledge Base 35
3.1.1 Ontologies 36
3.1.2 Building 37
3.1.3 Indexing 41
3.2 Context Model 42
3.2.1 Structural Context 42
3.2.2 Lexical Context 45
3.2.3 Context Transitions 46
3.3 Context-Based Search 49
3.3.1 Retrieval 50
3.3.2 Ranking 50

3.4 Context-Based Recommendation . . . 57
3.4.1 Retrieval . . . 57
3.4.2 Ranking . . . 59
3.5 Weight Learning . . . 60
3.6 Summary . . . 63

Chapter 4: Implementation . . . **65**
4.1 Architecture . . . 65
4.1.1 Data Layer . . . 65
4.1.2 Business Layer . . . 66
4.1.3 Presentation Layer . . . 67
4.2 Features . . . 68
4.2.1 Search . . . 68
4.2.2 Recommendation . . . 69
4.2.3 Monitor . . . 70

Chapter 5: Validation . . . **75**
5.1 Preliminary Study . . . 75
5.1.1 Context-Based Search . . . 78
5.1.2 Context-Based Recommendation . . . 81
5.2 Final Study . . . 85
5.2.1 Context Model . . . 85
5.2.2 Context-Based Search . . . 88
5.2.3 Context-Based Recommendation . . . 92
5.3 Discussion . . . 98
5.4 Limitations . . . 100

Chapter 6: Related Work . . . **103**
6.1 Context Awareness in Software Development . . . 103
6.2 Software Exploration . . . 107
6.2.1 Textual Approaches . . . 108
6.2.2 Static Approaches . . . 110
6.2.3 Textual/Static Approaches . . . 111
6.3 Retrieval in Software Reuse . . . 111
6.3.1 Lexical Retrieval . . . 112
6.3.2 Structural Retrieval . . . 113
6.3.3 Lexical/Structural Retrieval . . . 114
6.4 Software Project History . . . 115

Chapter 7: Conclusions . . . **117**
7.1 Contributions . . . 118
7.2 Future Work . . . 120
7.2.1 Knowledge Base . . . 120
7.2.2 Context Model . . . 122
7.2.3 Context-Based Search . . . 123
7.2.4 Context-Based Recommendation . . . 124
7.2.5 Weight Learning . . . 124
7.2.6 Application Domain . . . 124

References . . . **127**

List of Figures

2.1 Relations between the different types of ontologies, adapted from (Guarino, 1998). 9
2.2 The five fundamental categories for context information, adapted from (Zimmermann et al., 2007). 12
2.3 A screenshot of the Eclipse IDE. 25

3.1 The layered context model of the developer, crossing the different dimensions that comprise her/his work environment. 34
3.2 The conceptual architecture of our approach to context-based retrieval in software development. 35
3.3 The structural and lexical ontologies model. 36
3.4 Abstract representation of the process used to build the knowledge base. . . . 38
3.5 Example of a simplified AST generated for the source code of listing 3.1. . . . 39
3.6 Example of how the source code of listing 3.1 is represented in the structural and lexical ontologies. 40
3.7 Abstract representation of the process used to build the context model. . . . 42
3.8 Example of how the context model is derived from a set of interactions in the interaction timeline. 46
3.9 Example of how the context transition window is applied to the interaction timeline. 47
3.10 Abstract representation of the context-based search process. 50
3.11 Example of the source code elements retrieved for a given query. 51
3.12 Example of the structural paths between a set of retrieved structural elements (RE) and the elements in the structural context (CE). 53
3.13 Example of the lexical paths between the terms extracted form a set of retrieved structural elements (RE) and the terms in the lexical context (CT). 56
3.14 Abstract representation of the context-based recommendation process. . . . 57
3.15 Example of the retrieval and ranking using both the interest and the time based methods. 58

4.1 The architecture of the prototype implemented. 66
4.2 A screenshot of the prototype showing the search view (1) and the search window (2). 68
4.3 A screenshot of the prototype showing the recommendation view (1), the recommendation window (2), and the integration of recommendations in the search interfaces (3 and 4). 69
4.4 A screenshot of the monitor interface showing information about the structural context, including the list of existing context models (1), the list of events associated to the current context model (2), the list of structural elements (3), the list of structural relations (4), the list of events associated to the current structural element (5), and the evolution of the interest of the current structural element (6). 72

4.5 A screenshot of the monitor interface showing information about the lexical context, including the list of terms (1), the list of events associated to the current term (2), and the evolution of the interest of the current term (3). . 72
4.6 A screenshot of the monitor interface showing information about the structural ontology, including the number of structural elements (1), the number of structural relations (2), the evolution in the number of structural elements (3), and the evolution in the number of structural relations (4). . . 73
4.7 A screenshot of the monitor interface showing information about the lexical ontology, including the evolution in the number of terms (1), the number of lexical relations (2), and the evolution in the number of lexical relations (3). 73
4.8 A screenshot of the monitor interface showing information about the context-based search, including the number of selected search results (1), the distribution of the search weights (2), the evolution of the average rankings of selected search results (3), and the evolution of the search weights (4). . . . 74
4.9 A screenshot of the monitor interface showing information about the context-based recommendation, including the number of selected recommendations (1), the distribution of the recommendation weights (2), the evolution of the average rankings of selected recommendations (3), and the evolution of the recommendation weights (4). 74

5.1 Evolution of the search weights for developers #4, #15, #6 and #9, where the x-axis represents each weights update and the y-axis represents the value of the weights. 91
5.2 Evolution of the recommendation weights for developers #4, #10A, #6 and #21, where the x-axis represents each weights update and the y-axis represents the value of the weights. 97

List of Tables

2.1 The Recommender System for Software Engineering (RSSE) design dimensions, adapted from (Robillard et al., 2010). 31

3.1 Example of the terms used to index the source code elements of listing 3.1. 42
3.2 List of captured interactions, their description and interest variation. 43
3.3 Example of how a set of consecutive interactions affects the interest of a structural element. 44
3.4 Example of the term frequencies, inverse document frequencies, weights and scores computed for the query illustrated in figure 3.11. 52

5.1 The mean and confidence interval for the rankings of the selected search results, per component and experiment. 79
5.2 The weighted mean for the final weights of the context-based search, per component and experiment. 79
5.3 Questionnaire results for the context-based search, including the mean and the standard deviation, per group of developers. 80
5.4 Number of selected recommendations, per interface and group of developers. 82
5.5 The mean and confidence interval for the rankings of the selected recommendations, per component and group of developers. 82
5.6 The weighted mean for the final weights of the context-based recommendation, per component and group of developers. 83
5.7 Questionnaire results for the context-based recommendation, including the mean and the standard deviation, per group of developers. 84
5.8 The number of days of usage and average knowledge base sizes, per developer. 86
5.9 The average number of structural and lexical elements in the context model. 86
5.10 The percentage of times each relation appeared in the relations between added and existing context elements. 87
5.11 The statistical information collected about the context transition process. . 87
5.12 The mean and confidence interval for the rankings of the selected search results, per component. 89
5.13 Comparison between the rankings of the individual components for the selected search results. 89
5.14 The number of selected results, average rankings and final weights for the context-based search, per developer. 90
5.15 The weighted mean for the final weights of the context-based search, per component. 91
5.16 The questionnaire results for the context-based search, including mean and standard deviation. 93
5.17 The number, percentage and average rankings of newly accessed source code elements found in recommendations, per value of N. 94
5.18 Number of selected recommendations, per interface. 95

5.19 The mean and confidence interval for the rankings of the selected recommendations, per component. 95
5.20 The number of selected results, average rankings and final weights for the context-based recommendation, per developer. 96
5.21 The weighted mean for the final weights of the context-based recommendation, per component. 96
5.22 Questionnaire results for the context-based recommendation, including the mean and the standard deviation. 99

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CVS Concurrent Versioning System

DM Data Mining

IDE Integrated Development Environment

IE Information Extraction

IFT Information Foraging Theory

IR Information Retrieval

FCA Formal Concept Analysis

JDT Eclipse Java Development Tools

LDA Latent Dirichlet Allocation

LSA Latent Semantic Analysis

LSI Latent Semantic Indexing

NLP Natural Language Processing

RF Relevance Feedback

RS Recommender System

RSSE Recommender System for Software Engineering

SAN Spread Activation Network

SDE Software Development Environment

SDiC Software Development in Context

TF-IDF Term Frequency/Inverse Document Frequency

UI User Interface

VSM Vector Space Model

Chapter 1

Introduction

"I don't know anything, but I do know that everything is interesting if you go into it deeply enough."

Richard Feynman

Although there is an intuitive meaning for context in our minds, it remains an ambiguous concept that is difficult to define. Furthermore, the interest in the many roles of context comes from different fields, such as literature, philosophy, linguistics and computer science, with each field proposing its own view of context (Mostefaoui et al., 2004). The term context typically refers to the set of circumstances and facts that surround the center of interest, providing additional information and increasing understanding. The context-aware computing concept was first introduced by Schilit and Theimer (Schilit and Theimer, 1994), where they refer to context as *"location of use, the collection of nearby people and objects, as well as the changes to those objects over time"*. Similarly, Brown et al. (Brown et al., 1997) define context as location, identities of the people around the user, the time of day, season, temperature, etc. A more generic definition was provided by Dey and Abowd (Dey and Abowd, 2000), who define context as *"any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves"*. Additionally, they define a context-aware system as a system that *"provide relevant information and/or services to the user, where relevancy depends on the user's task"*.

Especially in software development, the context of a developer can be viewed as a rich and complex network of elements across different dimensions. Although software development (McCarthy and McCarthy, 2006) may include all the activities that result in a software product, from its conception to its realization, here we focus on the process of writing and maintaining the source code. This activity is usually conducted by developers in an Integrated Development Environment (IDE), which provide a set of tools aimed to help developers develop their work in an integrated workspace.

With the increasing dimension of software systems, software development projects have grown in complexity and size, as well as in the number of requirements and technologies involved (Robillard et al., 2010). During their work, software developers need to cope with a large amount of contextual information that is typically not captured and processed in order to enrich their work environment. With workspaces frequently comprising hundreds, or even thousands, of artifacts, they spend a considerable amount of time navigating the source code or searching for a specific source code artifact they need to work with (Murphy et al., 2006; Ko et al., 2006; Starke et al., 2009). This is especially true when developers

work on software maintenance and evolution tasks, which require them to locate a specific feature in the source code, or understand what source code artifacts may be relevant for their task. But, as the number of source code elements in the workspace of a developer increase, the need to switch between different elements becomes more frequent and more time is spent locating the needed elements.

With the aim of helping developers understand the source code structure and find what they need, modern IDEs provide several features for searching and navigating the source code. For instance, Eclipse[1], one of the most used IDEs for the Java programming language (Goth, 2005), provides at least seven different views for navigating the source code (Murphy et al., 2006) and at least eight different types of search options, including lexical and structural searches (Starke et al., 2009). Despite this number of tools, developers still spend a considerable amount of time navigating the source code structure. According to a study conducted by Ko et al. (Ko et al., 2006), about 35% of the working time of a developer is spent on searching and navigating the source code. As observed by Murphy et al. (Murphy et al., 2006), the navigation tools are among the most used views in Eclipse. Also, between the views dedicated to navigating the source code, the *Package Explorer* view, which allows to browse the entire source code structure in the form of a tree, was by far the most used, with 74% of selections, followed by the *Search* view, with 11% of selections.

With regard to search, Starke et al. (Starke et al., 2009) have observed that most of the times the searches performed by developers are too generic, leading to a high number of search results. Moreover, developers have difficulties in evaluating the relevance of search results, and tend to skim the usually big list of search results, seeking any clue of relevance. These searches are usually limited to the matching of specific patterns in the lines of code that comprise a software system. A good example of such an approach is grep[2], an Unix utility for searching plain text with regular expressions. But the pattern matching approaches have several shortcomings, requiring a direct correspondence between the pattern and the text in the source code. The limitations of these approaches have been largely surpassed by the research carried out in the field of Information Retrieval (IR), where efforts have been devoted to *"finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)"* (Manning et al., 2008). Although source code is of a structured nature, the individual source code artifacts can be viewed as individual documents, and their content can be represented by textual references, such as identifiers, comments and literals. The improvements pledged by the use of IR techniques, encouraged researchers using these techniques to help developers finding relevant source code for their current task (Marcus et al., 2004; Lukins et al., 2008; Gay et al., 2009; Shao and Smith, 2009). But, despite the fact that context is argued to improve the effectiveness of IR systems (Jones and Brown, 2004; Doan and Brézillon, 2004), as far as we know, none of the previous approaches have used the contextual information of the developer to improve the retrieval and ranking of relevant source code in the IDE.

Another form of delivering relevant source code artifacts to a developer is using a recommender system. These systems are currently used in a wide variety of domains to help users find relevant information, deal with information overload and provide personalized recommendations of very different kinds of items. The recommendation process is commonly dependent on estimating the utility of a specific item for a particular user (Adomavicius and Tuzhilin, 2005). Most of the approaches used in current recommender systems are mainly focused in estimating how relevant is an item to an user, ignoring any contextual information that could be used to improve the recommendation process. How-

[1] `http://www.eclipse.org/` (August 2012)
[2] `http://www.gnu.org/software/grep/` (August 2012)

ever, context-based recommendation systems are emerging, taking context into account when providing recommendations to the user (Adomavicius and Tuzhilin, 2011).

A recommender system for software engineering has been defined by Robbilard et al. (Robillard et al., 2010) as *"a software application that provides information items estimated to be valuable for a software engineering task in a given context"*. The increasing dimension and complexity of software development projects are fostering the development of such systems, which have been applied to very different tasks such as software reuse, expertise location, code comprehension, guided software changes, debugging, etc. The contextual information is of vital importance in these systems, serving as input for the recommendation process and determining the quality of its output (Robillard et al., 2010). A better description of the user context would allow for more focused and pertinent recommendations (Happel and Maalej, 2008).

Having identified the potential of recommender systems for software development, researchers studied ways of using contextual information, either implicit or explicit, to recommend source code artifacts that are potentially relevant for the current task of the developer. For instance, the history of interactions between the developers and the source code was used to identify navigational patterns, which allowed the recommendation of relevant artifacts given a current artifact (Singer et al., 2005; DeLine et al., 2005; McCarey et al., 2005). With the same objective, the information stored in the project memory of a software product was used to identify relationships between source code artifacts (Ying et al., 2004; Zimmermann et al., 2005; Cubranic et al., 2005). The context associated to the current task of the developer was used to help focus the information displayed in the IDE (Kersten and Murphy, 2006), to improve awareness, and facilitate the exploration of source code (Parnin and Gorg, 2006; Saul et al., 2007; Robillard, 2008; Piorkowski et al., 2012).

The research described in this thesis is focused on the development of context-based approaches to search and recommendation of source code in the workspace of the developer. The source code structure stored in the workspace of the developer is represented in a knowledge base. A context model represents the source code elements that are more relevant for the developer in a specific moment. These structures are then used to improve the retrieval and ranking of source code elements, such as classes, interfaces and methods, taking into account their relevance to the current context of the developer. In the remaining of this chapter, we state the main goals of our research, briefly describe our approach, refer to the main contributions of our work, and describe the structure of this thesis.

1.1 Research Goals

As we have observed in previous studies, developers spend a considerable amount of time navigating and searching for the source code elements needed for their tasks. Although current IDEs already provide a set of tools, aimed to help developers navigate and find what they need, these tools still suffer from some limitations that leave room for improvements. Especially, concerning the search of source code in the IDE, as far as we know, none of the existing approaches take into account the contextual information of the developer during the retrieval and ranking of search results, which could be used to improve the accuracy of source code search in the IDE. With regard to the recommendation, the use of contextual information is generally limited or based on information explicitly provided by the developer.

Being aware of this scenario, our objective was set on the development of a *context-based retrieval* approach for *software development*. This approach should take into account a *context model* of the developer, which should be used to improve the *search* and *rec-*

ommendation of source code that is being created or maintained by the developer on the *workspace* of an IDE. This generic objective can be decomposed in the following research goals:

- The definition of a *context model* of the developer capable of representing both the structural and lexical elements of the source code that are more relevant for the developer in a specific moment. This context model should be automatically created and continuously adapted to the current focus of attention of the developer.
- The development of a *context-based* approach to *search* and *recommendation* of relevant source code elements in the workspace of the developer. This approach should take into account the aforementioned context model, to retrieve and rank the source code elements according to their relevance to the developer.
- The development of a *prototype* to integrate the context-based search and recommendation approaches developed into an existing IDE. This prototype should be implemented taking into account the requirements of stability, performance and usability required to assure the validation of our approach in a real world scenario.

1.2 Approach

The approach we propose for achieving our research goals begin with the definition of a *knowledge base*. This knowledge base represents the source code structure stored in the workspace of the developer. The source code is represented from a structural and lexical perspectives, which are formalized using ontologies. A structural ontology is used to make explicit the different types of source code elements, as well as the structural relations that exist between them. While a lexical ontology represents the terms that comprise the identifiers of such source code elements, as well as the co-occurrence relations that exist between the terms.

The contextual information of the developer is modeled in the form of a *context model* that is grounded in the source code elements that are being manipulated by the developer. As in the knowledge base, this context model combines a structural and a lexical dimensions, which represent the source code elements, their structural relations and terms, that are more relevant for the developer in a specific moment in time. A context transition detection mechanism allows the context model to automatically adapt to the changes in the focus of attention of the developer.

The context model defined is used to improve the ranking of source code elements retrieved using a *context-based search* process. The retrieval is performed using the Vector Space Model (VSM) (Salton et al., 1975) of IR, by matching the terms in the identifiers of the source code elements with the terms contained in the search query. The retrieved elements are then ranked according to a retrieval, a structural and a lexical components. The retrieval component represents the ranking provided by the IR model, while the structural and lexical components represent the proximity of the search result to the context model of the developer. The contribution of these components to the ranking of search results is defined by a set of weights that are learned over time. The best combination of weights is inferred through an implicit feedback mechanism based on the search results selected by the developer.

The same context model was used to support the *context-based recommendation* of relevant source code elements to the developer. The recommendations are retrieved using the source code elements with higher interest and accessed more recently, that are represented in the context model. The ranking of recommendations takes into account an interest and a time components, representing the ranking obtained through the retrieval

process, as well as a structural and lexical components, which represent the proximity of the recommendation in relation to the context model. As in the context-based search process, the contribution of each one of these components is learned over time, through the analysis of the recommendations selected by the developer.

Finally, we have implemented a *prototype* that implements and integrates our approach in the Eclipse IDE, one of the most used IDEs for the Java programming language.

1.3 Contributions

The main contributions of this work are the use of a context model of the developer to improve the search and recommendation of source code in the IDE. Another important contribution is the development of a functional prototype, which provides access to context-based search and recommendation of source code in the Eclipse IDE. In a more systematic way, the contributions of this thesis can be summarized as follows:

- **Context Model.** The *context model* we have used in our approach introduces innovations that were not considered before. Although the context model used was inspired by previous work, it extends the existing model with a lexical perspective, which allowed us to explore the lexical relations between the source code elements, the same way that the structural relations were explored before. Additionally, we have also developed a *mechanism to automatically detect context transitions.* The aim of this mechanism is to detect changes in the focus of attention of the developer and reflect those changes in the context model.

- **Context-Based Search.** The context model developed was used to support an *approach to context-based search of source code in the IDE.* The search results retrieved are ranked according to the retrieval process and the context model of the developer.

- **Context-Based Recommendation.** The same context model was also used to support an *approach to context-based recommendation of source code in the IDE.* The elements in the context model are used to retrieve recommendations of relevant source code elements to the developer. These recommendations are than ranked according to their relevance in relation to the entire context model.

- **Learning.** A *learning mechanism* was used, so that the ranking of search results and recommendations could be adapted to the needs of the developer. This mechanism uses the rankings of the search results and recommendations, selected by developers, to favor the components that have a positive influence in their final ranking.

- **Prototype.** The context-based search and recommendation approaches developed were implemented and integrated in the Eclipse IDE, using a plugin named *Software Development in Context (SDiC).* This plugin can be easily installed in any Eclipse instance, providing access to context-based search and recommendation of source code in the IDE. The prototype is publicly available for download through the web site of the SDiC project (`http://sdic.dei.uc.pt`), representing also an important contribution of our research.

Finally, most of the work performed during the course of this thesis is published and was presented in international events, such as ECAI, RecSys, SEKE or ICSOFT (see a complete list in section 7.1).

1.4 Outline

The following chapters start with the theoretical background that supports the research developed under this thesis. Then, our approach and its different components are presented, as well as the prototype that implements and integrates our approach in the Eclipse IDE. The experiments performed to validate this approach are described next, along with a discussion of their results. Before concluding, we provide an overview on related work, comparing our work with other approaches. Finally, the thesis concludes with some final remarks, contributions and future work. A brief description of each one of the chapters is provided in the following paragraphs.

Chapter 2 provides an introduction to the *theoretical background* that supports the research developed under this thesis. We start with an introduction to ontologies, along with some of their classification methods, building methodologies and applications. Then, we provide a description of what context is and how it can be modeled. Next, we present an overview of the IR field, and continue to the field of Recommender System (RS). The chapter concludes with a perspective about software development, especially focused on the needs of developers while working on an IDE.

Chapter 3 presents the conceptual architecture and describes the individual components that embody the *approach* we have developed. After presenting an overview of the approach, we describe the knowledge base, which represents the source code structure, and the context model, which models the contextual information of the developer. Then, we explain how these structures are used to support the context-based search and recommendation processes. Finally, we present the learning mechanism that adapts the ranking of search results and recommendations by using the implicit feedback of the developer.

Chapter 4 describes the *prototype* that resulted from the implementation and integration of our approach in the Eclipse IDE. The architecture of the prototype is presented, with a description of the modules comprising the data, business and presentation layers. Then, we describe the main features provided to the developer, including the context-based search and recommendation interfaces, as well as a monitor interface that shows relevant information about the different modules.

Chapter 5 discusses the *experiments and results* that are in the basis of the validation of our research. The chapter starts with the description and results of a preliminary study, which was essential to identify some issues that needed to be addressed. A final study was then conducted and its results are presented next, including an evaluation of the context model and the context-based search and recommendation processes. The chapter concludes with a general discussion of the results obtained in the two studies.

Chapter 6 gives an overview of *related work*. We start by introducing some of the works that focus on context awareness in software development. Then, a set of works that tackled the problem of software exploration are presented. Next, we also include some works dedicated to the problem of software reuse. Finally, a set of works that explore the information extracted from a software project memory are described.

Chapter 7 concludes this thesis with some *final remarks*, highlighting its main *contributions* and providing several clues for *future work*.

Chapter 2

Background Knowledge

"If I have seen further it is by standing on the shoulders of giants."

Isaac Newton

This chapter provides an overview of the theoretical aspects that are in the basis of our work. We start by introducing the concept of ontology as a knowledge representation structure, its classification approaches, building process and applications. Then, context and its ambiguous nature are described, along with the approaches used for modeling contextual information. The Information Retrieval (IR) field, which provides the framework for textual search processes, is presented and described in relation to modeling approaches, text processing operations, query formulation, evaluation and context-awareness. The field of recommender systems, which is in the basis of any recommendation process, is also presented, including the classification of recommender systems, their evaluation and context integration paradigms. Finally, we introduce the field of software development, with a special focus on how developers search and collect relevant information for their needs.

2.1 Ontologies

An *ontology* is a powerful mechanism for representing knowledge and encoding its meaning, allowing the exchange of information that machines are able to process and concisely understand. Because of these characteristics, ontologies are one of the most important concepts present in the Semantic Web (Berners-Lee et al., 2001) architecture. There are various definitions of an ontology, going from a simple taxonomy to a strongly semantic and logic encoded conceptualization of the world. The term ontology came from the philosophy discipline, where it represents a branch dedicated to the study and description of existence and reality (Zuniga, 2001). Once the term was adopted in the computer science domain, it assumed a distinct role. A commonly cited definition is given by Gruber (Gruber, 1993):

> *"An ontology is an explicit specification of a conceptualization."*

Furthermore, Gruber defines a conceptualization as an abstract and simplified vision of the world that for some reason we want to represent. In a more refined definition, Guarino (Guarino, 1998) defines an ontology as:

> *"An engineering artifact, constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words."*

Both definitions take an ontology as an abstract conceptualization of a certain reality, and Guarino goes beyond that by saying it comprises a defined vocabulary and a set assumptions on the meaning of this vocabulary. Usually, the vocabulary definition is based on unary and binary predicates, called concepts and relations, respectively, and the assumptions are represented using first-order logic theory (Smullyan, 1968). Generally, an ontology contains an associated syntax, which represents the order and form of the symbols that comprise the ontology; some structure, that denotes the organization of the symbols and their relations; semantic information, containing the meaning of the symbols defining what they represent; and pragmatics, showing how the symbols can be used and their purpose (Maedche, 2002).

Some reasons that can be pointed out for the development of an ontology are described by Noy and McGuinness (Noy and McGuinness, 2001), which we enumerate:

- To share common understanding of the structure of information among people or software agents;
- To enable reuse of domain knowledge;
- To make domain assumptions explicit;
- To separate domain knowledge from the operational knowledge;
- To analyze domain knowledge.

Having tried to find a common definition of an ontology, we will now make some considerations about its classification approaches, development process and applications.

2.1.1 Classification

We can distinguish different types of ontologies. A classification system has been purposed by Guarino (Guarino, 1998), based on the level of generalization of the ontology (see figure 2.1). This classification is divided in four types of ontologies:

- *Top-Level Ontologies* describe general concepts that are independent from a specific problem or particular domain. It is reasonable to have top-level ontologies for large communities of users.
- *Domain Ontologies* describe the vocabulary related to a generic domain, through the specialization of concepts from top-level ontologies.
- *Task Ontologies* are similar to domain ontologies, but they are used in the domain of a specific task.
- *Application Ontologies* are the most specific ontologies, describing concepts that are a specialization of both domain ontologies and task ontologies. Generally, these concepts correspond to roles played by domain entities while performing a certain task.

In another classification mechanism, Heijst (van Heijst et al., 1997) goes beyond the subject of the conceptualization and classifies ontologies based on the structure of the conceptualization, defining three types of ontologies:

- *Terminological Ontologies* such as lexicons, defining the terms needed to represent the knowledge in the domain of discourse.

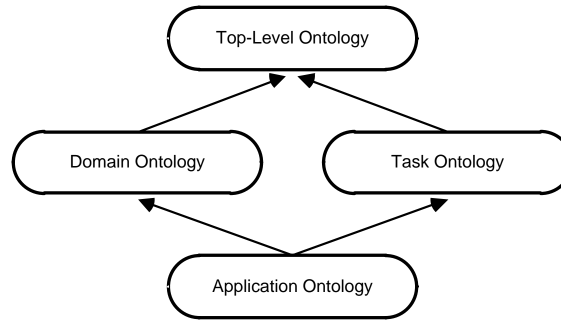

Figure 2.1: Relations between the different types of ontologies, adapted from (Guarino, 1998).

- *Information Ontologies* which specify the record structure of databases.
- *Knowledge Modeling Ontologies* that specify conceptualizations of knowledge, having a richer internal structure than information ontologies, and that are optimized for a particular use of the knowledge that they describe.

An ontology can also be viewed as a particular knowledge base, as it defines a structure and a shared vocabulary, which meaning is agreed and assumed to be always true by a community of users. A generic knowledge base generally also describes facts, being composed by an ontology and a "core" knowledge base that stores instances of the structures defined in the ontology (Guarino, 1998).

2.1.2 Building Process

Although there is more than one methodology for building ontologies, we will briefly describe one of the most used methodologies, developed by Noy and McGuinness (Noy and McGuinness, 2001). First of all, we must take in mind that there are always various alternatives for building an ontology, and that the one we choose will depend on the application we plan for the ontology being created. We need to choose the alternative that better reflects our reality, being at the same time the more intuitive, extensible and maintainable. According to Noy and McGuinness, the life-cycle for the development of an ontology comprises different steps:

1. *Determine the domain and scope of the ontology.* It is suggested to start the development of an ontology by defining its domain and scope. This comprises the definition of the domain that the ontology will cover, the purpose of the ontology, for what type of questions the information in the ontology must provide answers and who will use and maintain the ontology. This may change during the design process of an ontology, but it helps to limit the scope of the model.

2. *Consider reusing existing ontologies.* The reuse of ontologies can be done either for taking advantage of existing work, which can be done by refining or extending existing sources for our particular domain, or because it may be a requirement if our system must interact with other systems that use their own ontologies. There are a lot of ontologies available in electronic format and represented using standard languages that make their integration much easier. The Swoogle[1] (Ding et al., 2004) project is an example of a tool that helps reusing existing ontologies.

[1] `http://swoogle.umbc.edu/` (August 2012)

3. *Enumerate important terms in the ontology.* Before starting to define the classes and the corresponding class hierarchy, it is important to write down some terms closely related to the domain of the ontology. This helps to start defining the terms we want to talk about and their properties. In this phase we do not need to worry about relations or the overlap among terms.

4. *Define the classes and the class hierarchy.* This step and the following one are the most important and are also closely related. There are several approaches in developing a class hierarchy: a top-down development process that starts with the definition of the most general concepts in the domain and subsequent specialization of concepts; a bottom-up development process that starts with the definition of most specific classes, with subsequent grouping of these classes into more general concepts; and a combination development that is a combination of the last two approaches, starting by defining the more important concepts and ending up generalizing and specializing them properly.

5. *Define the properties of classes.* Once we have defined some classes, we have to describe the internal structure of concepts. There are several types of properties that can be defined, such as properties, parts and relationships.

6. *Define the restrictions on the properties.* Properties can have different restrictions, for instance referring to the value type, allowed values and cardinality of the values.

7. *Create instances.* The last step is the definition of the classes. This is done by selecting the class we want to instantiate, creating an individual instance of that class and filling the properties values.

This is an iterative approach and we can go through the various phases several times, starting with a rough version of the ontology and filling in the details progressively to evolve this ontology.

One of the biggest challenges of Semantic Web is to efficiently and effectively construct ontologies, since they are the core representational medium for knowledge. When possible, this should be done through some automated process, which makes things more complex. In their work, Brewster et al. (Brewster et al., 2005) discuss the foundations on which automated ontology construction should build, as well as a set of functions that an ontology should fulfill. They argue that these foundations and the available resources for ontology construction are highly problematic. Furthermore, there are different requirements in ontology construction that can be contradictory and impossible to be achieved simultaneously.

2.1.3 Applications

Compared to syntactic standards, ontologies not only provide a common representation and structure, but also help to reach a common understanding of the meaning of terms. These characteristics make ontologies a privileged mean to support semantic interoperability, which is a key factor in various applications (Maedche, 2002; Stuckenschmidt and van Harmelen, 2005):

- *Semantic Web* is a domain where ontologies are a key concept. As referred before, ontologies provide mechanisms that structure and semantically describe knowledge, which is essential to make it machine understandable. Some examples of the use of ontologies in Semantic Web prototypes are described by Staab, et al, (Staab and Maedche, 2001), introducing knowledge portals for different application scenarios.

- *Natural Language Processing* generally requires the integration of many knowledge sources, where the domain knowledge represented as an ontology is a key factor for the correct understanding of texts.
- *Information Retrieval* techniques require a shared vocabulary between the user and the source information, preventing the need of a specific encoding and reducing the ambiguity of the word matching, which makes the retrieval more efficient and precise.
- *Knowledge Management*, which main roles include acquiring, maintaining and accessing knowledge of an organization, with this knowledge being mainly stored in the form of documents. The mechanisms provided by ontologies allows one to move from a document oriented approach to a knowledge resource approach, where every structured resource is concisely described and interconnected in a flexible way.
- *e-Business*, which needs automation mechanisms, thus requiring a mean for common understanding and interpretation of terms, given by ontologies, assuring the desired interoperability and information integration.

2.2 Context

The term *context* has an intuitive meaning for humans, but due to this intuitive connotation it remains vague and generalist (Mostefaoui et al., 2004). Furthermore, the interest in the many roles of context comes from different fields, such as literature, philosophy, linguistics and computer science, with each field proposing its own view of context. Referring to the computer science field, it is of particular interest for areas such as artificial intelligence, context-aware systems and IR. Especially in the IR field, the use of context is argued to improve the effectiveness of IR systems (Jones and Brown, 2004). But the use of context to improve IR is a big challenge and most of these systems retrieve and rank information based only on queries and document collections, ignoring the full potential of the user's context.

2.2.1 What is context?

The term *context* typically refers to the set of circumstances and facts that surround the center of interest, providing additional information and increasing understanding. According to the American Heritage Dictionary[2] of the English Language, the term context has two meanings:

> 1. *"The part of a text or statement that surrounds a particular word or passage and determines its meaning."*
> 2. *"The circumstances in which an event occurs; a setting."*

The first definition is closely related to linguistics, while the second one, more generalist, can be applied in other fields. A large number of definitions for the terms context and context-aware have been proposed. The term context-aware computing was first introduced by Schilit and Theimer (Schilit and Theimer, 1994), where they refer to context as:

> *"... location of use, the collection of nearby people and objects, as well as the changes to those objects over time."*

[2] `http://www.ahdictionary.com/` (August 2012)

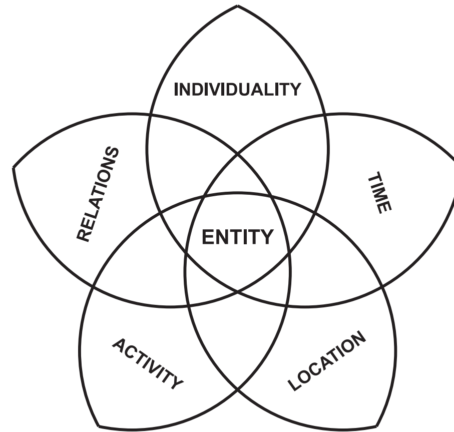

Figure 2.2: The five fundamental categories for context information, adapted from (Zimmermann et al., 2007).

In a similar way, Brown et al. (Brown et al., 1997) define context as location, identities of the people around the user, the time of day, season, temperature, etc. In a more generic definition, Dey and Abowd (Dey and Abowd, 2000) define context as follows:

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves."*

As stated by Mena et al. (Mena et al., 2007), it is clear the importance of context for modeling human activities (reasoning, perception, language comprehension, etc), which is also true in social sciences and computing. But a generic context definition is hard to achieve, as context assumes different definitions and characteristics according to the domain where it is used. Nevertheless, there are a set of invariant characteristics in the different definitions of context: context always relates to an entity; is used to solve a problem; depends on the domain of use and time; and is evolutionary because it relates to a dynamic process in a dynamic environment.

2.2.2 Context Modeling

In an attempt to overcome the generality of common definitions of the term context, an operational definition of context is proposed by Zimmerman et al. (Zimmermann et al., 2007). This definition comprises three canonical parts: a definition in general terms, a formal definition regarding the appearance of context, and an operational definition characterizing the use of context and its dynamics. The general definition is based on the definition provided by Dey et al. (Dey and Abowd, 2000), which defines context as *"any information that can be used to characterize the situation of an entity"*. As a formal extension to this definition, Zimmerman et al. state that the elements used on the description of context information can be grouped in five categories: individuality, activity, location, time and relations (see figure 2.2). The grouping of contextual information in these five clusters is considered vital for a pragmatic approach, facilitating the engineering and management of a context model for context-aware systems.

The *individuality context* comprises the contextual information about the entity the context is bound to. It contains information related to anything that can be observed about an entity. This context information can be of four types:

- *Natural Entity Context.* This category comprises the characteristics of all natural living and non-living things that are not result of human intervention.

- *Human Entity Context.* This category covers the characteristics of human beings. The General User Model Ontology (GUMO) (Heckmann et al., 2005) is seen as a potential source of characteristics to take into account.

- *Artificial Entity Context.* This category contains information about products or phenomena that result from human actions or technical processes.

- *Group Entity Context.* This category comprises information about groups of entities which share characteristics, interact with one another or have established relations. It is important to take into account these groups when capturing characteristics that emerge only when entities are grouped together.

The *time context* has an important role in context information, as most of its elements are related over the temporal dimension. The storage of contextual information over time allows the analysis of the interaction history and behavior habits of users, which can be used to predict future contexts.

The *location context* gains relevance as portable computing devices become more relevant and humans start to move in an ubiquitous computing environment. The location models classify the physical or virtual location of an entity, as well as other spatial information, such as speed and orientation. These models can be split into quantitative (geometric) and qualitative (symbolic) models. The quantitative models refer to geometric coordinates and the qualitative models refer to elements such as buildings, rooms, streets, countries, etc.

The *activity context* covers the information about the activities performed by the user, which determine to a great extent its current needs. This information describes what the user wants to achieve and how, and can be defined by means of explicit goals, tasks and actions.

The *relations context* comprises information about the relations between an entity and other entities, which can be persons, things, devices, services or other entities. Because the set of possible relation types is large, a clustering of relations regarding the types of entities involved is proposed, splitting the relations into three types:

- *Social Relations.* This category describes social aspects of the entity context, usually, interpersonal relations such as social associations, connections or affiliations between two or more people. This information can be used as a basis for deriving patterns in behavior and groups of people with similar interests, goals or levels of knowledge.

- *Functional Relations.* This category comprises relations where an entity makes use of other entity, for a certain purpose and a certain effect.

- *Composition Relations.* This category includes relations between the whole and its parts. In special, the aggregation relations, which imply that the parts will not exist anymore if the containing object is removed.

The operational extension to the context definition, proposed by Zimmerman et al. (Zimmermann et al., 2007), addresses its use and its dynamic properties: the transitions between contexts of one entity and the sharing of contexts among several entities. Particularly, human entities change contexts and two consecutive contexts are never exactly the same.

The transition between contexts can be defined as a variation of approximation, a change of focus or a shift of attention. A variation of approximation occurs when the

contextual knowledge represented by a context experiences a specialization or a generalization. The change of focus refers to a change in the reachability or accessibility of specific elements of the context description in a specific situation. The shift of attention relates to a change of the focus of attention on specific aspects of the contextual knowledge, which is largely determined by the type and amount of knowledge required for the processing of the current activity.

The sharing of context emerges when the contexts of two entities overlap and parts of the context information become similar and shared. The emergence and exploitation of context is described through the establishment of relations, adjusting of shared contexts and exploiting of relations. The establishment of relations occurs when entities converge spatially or temporally, this proximity enables them to start responding to each other. The adjustment of contexts is necessary because the participants in an interaction need to share the same understanding or interpretation of the meaning of a context description. Finally, the larger the shared context between two interacting parties, the more an exploiting of relations is facilitated, since they better understand each other.

2.3 Information Retrieval

The field of *Information Retrieval (IR)* is very broad, but an academic definition is given by Manning et al. (Manning et al., 2008):

> *"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."*

Although IR has been seen as an activity performed only by professional searchers, current advances in the World Wide Web allow millions of ordinary users to perform IR through web search engines such as Google[3], Yahoo![4] or Bing[5]. IR is becoming the dominant form of information access and IR applications continue to grow and expand to larger archives and new computing environments. The IR field is not limited to the definition provided above, providing also support for browsing and filtering document collections. The IR techniques are also used to perform clustering tasks, by identifying groups of documents based on their contents, and classification tasks, by assigning documents to one, or more, previously provided topics.

A typical IR process starts with the user expressing an information need, usually represented as a *query*, which in its simplest form can be composed by one or more *terms*. The query is processed by the IR system and a set of *documents*, from a *collection*, that match the query are retrieved. The match between the query and the documents in the collection requires that the same *text processing* operations are applied to both the query and the document. The retrieved documents are ranked according to their *relevance* to the query. When examining the retrieved documents, the user may identify those that are more interesting and provide *feedback* to help refining the retrieval process.

Here, we pretend to provide only an overview of the IR field, with a special focus in the areas that are of more relevance to this work. We start with a description of the most important IR models, which provide the grounding framework for an IR system. Then, we describe some of the text processing operations that are commonly used in IR. The query formulation process is also addressed, giving special attention to the different types of

[3]`http://www.google.com/` (August 2012)
[4]`http://www.yahoo.com/` (August 2012)
[5]`http://www.bing.com/` (August 2012)

queries available and the mechanisms that provide aid during the query formulation process. Next, the most relevant measures for evaluating IR systems are presented. Finally, we provide some insights of how context has been exploited to improve the IR process.

2.3.1 Modeling

A central problem of IR systems is to predict which documents are relevant, or not, in relation to a query. Such systems usually answer this problem by providing a list of documents ordered by a estimation of their relevance to the query. Thus, ranking algorithms are an essential part of IR systems, so that different interpretations of document relevance result in different IR models. As proposed by Baeza-Yates and Ribeiro-Neto (Baeza-Yates and Ribeiro-Neto, 1999), an IR model is characterized by:

- A set of representations for documents in a collection;
- A set of representations for queries;
- A framework for modeling document representations, queries, and their relationships;
- A ranking function that associates a real number between a query and a document representation.

The classical IR models describe documents and queries as sets of *index terms*, generally nouns, that represent the document contents. The importance of a term when used to describe the contents of a document is usually represented as a *weight*. These models have been categorized as *set-theoretic*, *algebraic*, and *probabilistic*. Following, we provide a brief description of each category, as presented in (Baeza-Yates and Ribeiro-Neto, 1999). The interested reader may find more information about the various IR models in the literature (Baeza-Yates and Ribeiro-Neto, 1999; Manning et al., 2008; Büttcher et al., 2010).

Set-theoretic Models

The *set-theoretic models* are based on set theory and Boolean algebra. These models are grounded on the original *Boolean model*, which interprets the relevancy of a term in a binary fashion, thus the weight of a term is 1 when the term is present in the document, or 0 otherwise. A query is a Boolean expression, comprising a set of terms connected by Boolean operators, such as `and`, `or` and `not`. A document is considered relevant or non-relevant, whether the terms in the document satisfy the given query or not, and partial matching is not supported. The simplicity of the model is an advantage, but its limitations usually lead to the retrieval of too many or too few documents.

As an attempt to overcome the limitations of the Boolean model, some derivative models have been proposed over the years. One of these models is the *extended Boolean model*, proposed by Salton et al. (Salton et al., 1983), which combines the characteristics of the Vector Space Model (Salton et al., 1975) with Boolean algebra, to introduce term weighting and partial matching into the classic Boolean model. Another group of derivative models are categorized as *fuzzy* retrieval models (Ogawa et al., 1991), which combine the extended Boolean model with properties of the fuzzy set theory.

Algebraic Models

The *algebraic models* are based in representations of documents and queries in the form of vectors and compute similarities using algebra. The original algebraic model, the *Vector*

Space Model (VSM), was proposed by Salton et al. (Salton et al., 1975) to overcome the limitations of the Boolean model, providing a framework for non-binary term weights and partial matching. The documents and queries are represented as t-dimensional vectors of term weights. The degree of similarity between a document and a given query is computed as a correlation of their associated vectors. This way, the vector model is able to retrieve documents which partially match the query, and rank the retrieved documents according to their degree of similarity to the query. The advantages of the vector model rely on its term weighting approach, partial matching support and ranking of documents according to a degree of similarity to the query. On the other hand, assuming that terms are mutually independent can be seen as a drawback, even if that using term dependencies is not a trivial task.

The degree of similarity between a document and a given query is usually computed using the *cosine similarity measure*, which represents the cosine of the angle between the two vectors. This value is computed using linear algebra calculations, and given that all the components of the two vectors are non-negative, the value of the cosine similarity measure ranges from 0 to 1, for angles between 0° and 90°, respectively. As shown in equation 2.1, the cosine similarity between a document vector $\vec{d}$ and a query vector $\vec{q}$ is represented as the dot product of their vectors normalized to unit length.

$$sim(\vec{d}, \vec{q}) = \frac{\vec{d}}{\mid \vec{d} \mid} \cdot \frac{\vec{q}}{\mid \vec{q} \mid} \tag{2.1}$$

The term weights can be computed using various methods that have been proposed and evaluated over the years. As originally proposed by Salton et al. (Salton and Buckley, 1988), the term weights are commonly computed using the *Term Frequency/Inverse Document Frequency (TF-IDF)*. The weights in a document vector represent the product of a function of the term frequency (TF) and a function of inverse document frequency (IDF). The weights in a query vector are usually represented using only a function of the term frequency within the query. The idea behind term frequency is that the more frequent a term is, in a document or query, the more relevant it is, in relation to that document or query. The term frequency for a term t in document d is usually computed using equation 2.2, where f_{td} is the frequency of the term in the document.

$$tf(t, d) = log\,(f_{td}) + 1 \tag{2.2}$$

The inverse document frequency, on the other hand, is based on the assumption that terms appearing in many documents are less relevant than terms appearing in few documents, and is computed using equation 2.3, where D is the total number of documents in the collection and df_t is the number of documents containing the term t.

$$idf(t) = log\left(\frac{D}{df_t}\right) \tag{2.3}$$

Finally, the weight w for a term t in document d is given by the product of $tf(t, d)$ with $idf(t)$, as shown in equation 2.4.

$$w(t, d) = (log\,(f_{td}) + 1) \times log\left(\frac{D}{df_t}\right) \tag{2.4}$$

Some alternative algebraic models have been proposed to overcome some of the limitations of the classical vector space model. The *generalized vector space model* (Wong et al., 1985) introduces term correlations into the original model. The *Latent Semantic Indexing (LSI)* model (Deerwester et al., 1990), also known as *Latent Semantic Analysis (LSA)* was proposed to map documents and queries to a lower dimensional space, composed of

concepts. Another alternative algebraic modes is the *neural network model* (Wilkinson and Hingston, 1991), which takes advantage of the pattern matching characteristics of neural networks to match and compute the similarity between queries and documents.

Probabilistic Models

As its name suggests, the *probabilistic model*, introduced by Robertson and Jones (Robertson and Jones, 1976), addresses the IR problem from a probabilistic perspective. This model is based on the assumption that given a user query and a document, it is possible to estimate the probability that the user will find the document interesting. This probability depends only on the query and document representations. The subset of the documents that the user would like to obtain for a given query, which maximize the probability of relevance to the user, is considered an ideal answer set. A document is predicted to be relevant if it is contained in this ideal answer set, or non-relevant otherwise. The model assigns each document a measure of similarity to a given query, which represents the odds of the document being relevant to that query. The documents are ranked according to the odds of relevance in relation to the query. The advantage of probabilistic models is that documents are ranked according to a probability of their relevancy, on the other hand they require an initial separation of the documents in relevant and non-relevant sets, ignore term frequencies and assume that terms are mutually independent.

A set of alternative probabilistic models are based on Bayesian networks (Pearl, 1988), including the *inference network model* (Turtle and Croft, 1990, 1991) and the *belief network model* (Ribeiro and Muntz, 1996). These models use the formalisms associated to Bayesian networks to model the probability of a document being relevant for a given query.

2.3.2 Text Processing

As discussed before, in IR, terms are used to represent the contents of a document. We have also described that terms are not equally significant when used to represent the contents of a document. This way, it is common to pre-process the text of a document to determine which terms should be used to index that document. According to Baeza-Yates and Ribeiro-Neto (Baeza-Yates and Ribeiro-Neto, 1999), this process can be divided into five main text operations: lexical analysis, elimination of stop-words, stemming, selection of index terms, and construction of the term categorization structures.

The *lexical analysis* is the process of transforming a stream of characters into a stream of tokens, or words. Although it may seem a simple procedure, it involves a careful analysis of some aspects, such as digits, hyphens, punctuation marks, and the case of letters.

The *stopwords removal* process has the objective of filtering words that are not relevant as index terms. A stopwords list includes words that appear too frequently in a collection of documents, such as articles, prepositions and conjunctions, which are not good discriminators and become useless to the retrieval process. The elimination of these frequent words has the advantage of reducing the size of the indexing structure, but may also reduce performance due to some information loss.

Taking into account that words may assume very different forms, typically conveyed by plurals, gerund forms, and past tense suffixes, it would be difficult to match a term in the query with the different forms in which it may appear in a document. The *stemming* technique can be used to overcome this problem, by reducing the various variants of a root word to the same concept. This process also reduces the size of the indexing structure, because the number of different index terms in the structure is largely reduced.

When creating a representation of a document, all the words, or a subset of them, can be used as *index terms*. When only a subset of the words contained in the document is

used to index the document, a criteria must be defined to determine which terms should be selected. Some of the approaches used for index terms selection include using only nouns, based on the intuition that nouns carry the semantics of the document. Sometimes, nouns are combined in groups to represent a specific concept, thus some approaches use groups of nouns as index terms.

The terms used to index documents may be organized in a more complex structure, such as a *thesaurus* (Roget, 1852). In its simplest form, a thesaurus of words comprises a precompiled list of important words in a given domain, along with a list of related words for each word in the list. This structure provides a standard vocabulary for indexing and searching, helps locating terms for query reformulation and provides a classified hierarchy that can be used to narrow or broaden the query according to the needs of the user.

2.3.3 Query Formulation

The information need of a user using an IR system is generally represented in the form of a query. The type of query provided to an IR system is dependent on the underlying IR model of that system, thus the various kinds of queries can be categorized in different groups. As described by Baeza-Yates and Ribeiro-Neto (Baeza-Yates and Ribeiro-Neto, 1999), queries can be classified as *keyword-based*, *pattern-based* or *structural-based*.

The *keyword-based* queries comprise words and, in some cases, a combination of operations over several words. The single-word queries are the most simple, containing only one word. The multiple-word queries are composed of more that one word and can be further divided in context, Boolean or natural language queries. The context queries allow to search for words in a given context, either in the form of a phrase, by matching the words in the query as sequence of words in the document, or by matching the words in the query within a maximum allowed distance in the document. The Boolean queries combine keyword queries with Boolean operators, such as `or`, `and`, `but`, etc., requiring the system to retrieve only the documents that satisfy the restrictions expressed in the query. The natural language queries assume that a query is a simply enumeration of words and context queries, eliminating the need to explicitly use Boolean operators.

The *pattern matching* queries are based on patterns, which can be defined as a set of syntactic features that must occur in a text segment. The complexity of a pattern range from a simple set of words to complex regular expressions, allowing the user to express prefixes, suffixes, substrings, ranges, error handling, unions, concatenations, repetitions, classes of characters, conditional expressions, etc.

The *structural* queries take advantage of the structure of text collections, allowing the user to query such texts based on their structure. They integrate both the contents and structure of documents in the query, providing a more powerful way of expressing the information needs of the user.

The different types of queries described provide a broad range of options to describe the information needs of the user. But, sometimes, it is difficult for the user to formulate a query suitable to obtain the best results in the IR process. This problem has been addressed by various approaches that try to improve the IR process by helping the user formulating, or reformulating, the query. These approaches can be categorized as *local methods* and *global methods*. The global methods, also known as *query expansion* methods, include techniques for expanding, or reformulating, a query using information derived from the entire document collection. The objective is to complement the initial query with terms that are synonymous or related with the original terms in the query. The related terms are usually derived from structured indexing structures, such as thesaurus.

The local methods rely on the initial set of documents retrieved by a query to achieve the same objective. The most representative is *relevance feedback* (Rocchio, 1971), one of

the most used and effective ways of improving retrieval performance in IR systems based on algebraic models. The basic idea behind relevance feedback is to use the feedback of the user to iteratively refine the query. The process starts with the user submitting a query to the IR system, which retrieves a set of documents in response to that query. Then, the user is required to specify which documents are considered relevant to satisfy the initial information need. The query is reformulated and weighted taking into account the distribution of terms in relevant and non-relevant documents, as defined by the user. The new query is submitted to the IR system, which retrieves a new set of documents. The process can be repeated until the information needs of the user have been satisfied. This process relies on the *explicit feedback* of the user, other forms of relevance feedback are *implicit relevance feedback* and *pseudo relevance feedback*. The implicit relevance feedback method infers the user feedback through the analysis of the user behaviour. The pseudo relevance feedback method assumes that a predefined number of the topmost ranked documents are relevant for the user, and executes the relevance feedback process using these documents, even before the user receives the initially returned documents.

2.3.4 Evaluation

Traditional evaluation of IR systems is based on the assumption that given an information need, represented by a query, every document in a collection can be classified as relevant or non-relevant, in respect to that query (Baeza-Yates and Ribeiro-Neto, 1999). Based on these assumptions, several evaluation measures have been proposed. The oldest and most used measures in IR are *recall* and *precision*. The *recall* measure quantifies the quantity of relevant search results that the system was able to retrieve, and is computed as the fraction of the relevant documents that has been retrieved (see equation 2.5). As we can see, it would be easy for a system to obtain a maximum score for *recall* by retrieving all the documents in the collection.

$$recall = \frac{\mid \{RelevantDocuments\} \bigcap \{RetrievedDocuments\} \mid}{\mid \{RelevantDocuments\} \mid} \tag{2.5}$$

The *precision* measure, on the other hand, quantifies the quality of the search results retrieved by the system, and is computed as the fraction of the retrieved documents that is relevant (see equation 2.6). Generally, *precision* takes into account all the documents that were retrieved, but it can also be computed at different cut-off levels, considering only the topmost results returned.

$$precision = \frac{\mid \{RelevantDocuments\} \bigcap \{RetrievedDocuments\} \mid}{\mid \{RetrievedDocuments\} \mid} \tag{2.6}$$

The *harmonic mean*, or *f-measure*, has been proposed to combine *recall* and *precision* in a single measure, providing a weighted average of both (see equation 2.7). It assumes a high value only when both *recall* and *precision* are high, and has been used to find the best compromise between the two measures.

$$f\text{-}measure = 2 \times \frac{recall \times precision}{recall + precision} \tag{2.7}$$

Several other measures have been proposed, which might be of interest in different situations. Here we have presented only the most relevant, please refer to the literature for a detailed description of other measures (Baeza-Yates and Ribeiro-Neto, 1999; Manning et al., 2008; Büttcher et al., 2010).

2.3.5 Context-Aware Information Retrieval

The growth observed in IR motivates the research for better search technologies, and the use of context is argued to improve the effectiveness of IR systems (Jones and Brown, 2004). Users of IR systems work in a personal and physical context, on the other hand the documents they search often relate to specific contexts. Another fact that reinforces the importance of context in IR relates to the developments in mobile and wireless computing, where the physical environment of the user can be seen as rich source of contextual information. The use of context to improve information retrieval is a big challenge and most IR systems retrieve and rank information based only on queries and document collections, ignoring the user interests and actual context. But, although the full potential of the modeling of context, in which the user performs the search, has been relegated to background importance in IR, it is already present in established techniques such as Relevance Feedback (RF) (Rocchio, 1971).

In their work, Doan and Brézillon (Doan and Brézillon, 2004) consider that one way to improve the efficiency of IR systems is to make explicit the context the query belongs to. The relevancy of the responses given by IR systems varies from user to user and is dependent from their contextual spaces. This contextual information is seen to be linked to the terms in the query, the user-profile, the system itself and the interactions between the user and the system. Also, context information can be explicit, when it is collected directly from existing information or introduced explicitly in the search process, or implicit, for instance when it is inferred from the user interactions with the system. They propose that the contextual information of the user can be build upon different sources: user profile context, IR system context, document context, and user/system interaction. The user profile represents the user preferences and is progressively filled in, often through a relevance feedback mechanism. The IR system context depends on the system itself, especially on the collected resources, the indexing algorithm, the matching algorithm, the query language and the display of results. The document context is typically related with the way documents are linked together, for instance using hypertext links. The user/system interaction can be seen, for instance, as a dialog, with the user expressing an information need, in which every query/response step can encode relevant information that belongs to that specific context.

During the last years, several researchers have approached the use of contextual information in IR. One of the first approaches for personalizing web search was proposed by Pitkow et al. (Pitkow et al., 2002), using a user model based on the navigation history of the user. In the following years, several works have explored the navigation history (Gauch et al., 2003; Sugiyama et al., 2004; Matthijs and Radlinski, 2011) and the search history (Liu et al., 2004; Speretta and Gauch, 2005; Tan et al., 2006; Kotov et al., 2011; Sontag et al., 2012) of the user to personalize web search. Other works have combined information stored in the desktop of the users with their browsing history to personalize web search results (Teevan et al., 2005; Chirita et al., 2006). In most of these approaches, the user profile is automatically built and used implicitly in the search process, but some approaches rely on explicit user profiles (Chirita et al., 2005; Ma et al., 2007), for instance requiring the user to select a set of interest topics in the Open Directory Project[6] catalog. While most of the research on personalized IR focuses on long-term models for representing the user interests, some approaches have explored the contextual information available within a search session to achieve personalization. These approaches usually make use of the queries and clicks performed during a search session to improve the ranking of search results (Shen et al., 2005; Daoud et al., 2009; Xiang et al., 2010), suggest new queries (Cao et al., 2008) or perform query disambiguation (Mihalkova and Mooney, 2009).

[6] `http://www.dmoz.org/` (August 2012)

2.4 Recommender Systems

A broad range of different *recommender systems* are currently used in a wide variety of domains to help users find relevant information, deal with information overload and provide personalized recommendations of very different kinds of items. The recommendation process is commonly dependent on estimating the *utility* of a specific item for a particular user (Adomavicius and Tuzhilin, 2005). The utility of an item to a user is usually represented by a *rating*, which can be explicitly provided by the user. The problem is that a recommender system rarely knows the rating for every user/item pair, thus the unknown ratings must be *estimated*. Based on the estimated ratings, a recommender system is able to *recommend* a set of useful items for a user, ordered by their estimated ratings, or even a set of relevant users for an item. Alternatively, instead of estimating the exact rating for a user/item pair, some recommender systems predict only the relative preferences of the user, what is known as *preference-based filtering* (Cohen et al., 1999).

Having the problem of rating estimation as a core issue, recommender systems are usually classified according to their approach for estimating ratings. In the following section we provide an overview of the three main categories of recommender systems. Then, we describe some of the approaches used for evaluating recommender systems. Finally, we discuss the use of context in recommender systems.

2.4.1 Classification

According to their approach for estimating the rating of a item for a user, recommender systems are classified as *collaborative*, *content-based* or *hybrid*. Following, we provide a brief description of each one of these categories, as provided by Adomavicius and Tuzhilin (Adomavicius and Tuzhilin, 2005).

Collaborative

The *collaborative* recommender systems, or colaborative filtering systems, predict the utility of an item for a user based on the items previously rated by other users (Resnick et al., 1994). These approaches are based on the assumption that items rated as useful for an user may be considered useful for other users that share the same preferences. According to Breese et al. (Breese et al., 1998), collaborative recommendations can be classified as *memory-based* (or heuristic-based) and *model-based*. The memory-based approaches use heuristics to compute rate predictions, based on a collection of previously rated items, while model-based approaches use the collection of rated items to learn a model that is used to predict ratings. Because they are based only on the ratings provided by users, collaborative recommender systems are able to recommend any type of items, independently of their contents. However, some of these systems have some limitations, such as the *cold-start* problem (Schein et al., 2002) and rating *sparsity* (Papagelis et al., 2005). The cold-start problem relates to the fact that the system requires a minimum number of previously rated items before being able to make accurate recommendations. Also, the number of previously rated items in the system is frequently very small compared to the number of ratings that must be predicted, this rating sparsity problem may lead to a poor performance of the system.

Content-Based

The *content-based* recommendation approaches estimate the utility of an item for a user through the utility assigned by the user to other items that are similar to that item (Basu et al., 1998). These approaches assume that items can be estimated useful for a user if they

are similar to other items that were rated useful for that user. An item is usually characterized by an item profile, composed by a set of features extracted from its content, that is used to evaluate when it should be recommended. These approaches were initially derived from IR (Baeza-Yates and Ribeiro-Neto, 1999) and Information Extraction (IE) (Belkin and Croft, 1992) research, where traditional information retrieval systems were improved by using information about the user preferences. Due to their origins, the content-based recommendation approaches are mostly used for recommending text-based items. These items are characterized by keywords and their similarity is often computed using term weighting approaches, such as TF-IDF (Salton and Buckley, 1988). Besides the heuristic approaches based on IR, other techniques for content-based recommendation base their utility function on a model learned from the data, using statistical learning and several machine learning techniques (Mitchell, 1997). Because the content-based approaches are only based on the contents of the items being recommended, they are highly dependent on the number and quality of features associated to the items. Another problem associated to these approaches is *over-specialization*, which causes the system to always recommend items similar to what is already known to the user (Balabanović and Shoham, 1997).

Hybrid

With the objective of overcoming some of the limitations of both collaborative and content-based approaches, some recommender systems combine the two and use an *hybrid* approach (Burke, 2002). The collaborative and content-based approaches may be combined in different manners, for instance by implementing the two approaches independently and combining their predictions, incorporating content-based characteristics in a collaborative approach, or vice-versa, or building an unified model that incorporates characteristics of the two approaches.

2.4.2 Evaluation

As discussed in (Jannach et al., 2011), the evaluation of a recommender system is essential for determining the accuracy of the system or evaluate how it compares with previous approaches. However, it is argued that the traditional approaches used for evaluating such systems have several limitations, or that the quality of a recommender system can not be adequately measured because there are too many objective functions. Nevertheless, these systems have been traditionally evaluated using offline experiments based on a data set, for instance containing an history of transactions.

The performance of a recommender system is usually measured using a set of metrics (Jannach et al., 2011). Concerning the *accuracy of predictions*, the *mean absolute error* (MAE) is the most used metric. As shown in equation 2.8, it is used to compute the average deviation between predicted ratings (p_i) and the real ratings (r_i).

$$MAE = \frac{\sum_{i=1}^{n} \mid p_i - r_i \mid}{n} \tag{2.8}$$

The relevancy of the recommendations given to the user by the recommender system, or the *accuracy of classifications*, is usually computed using the *precision*, *recall* and *f-measure* metrics, which were already presented in the context of IR (see section 2.3.4).

The rankings of the recommendations provide a higher level of granularity when evaluating a recommender system. To evaluate the *accuracy of ranks*, Breese et al. (Breese et al., 1998) proposed a metric based on the assumption that the lower the ranking of an item the lower is its utility, because items at lower rankings are likely to be ignored by the user. The rank score for user u is given by equation 2.9, where α represents the half-life of

utilities, $hits_u$ are the recommendations selected by the user and $rank(i)$ is the position of item i in recommendations.

$$RankScore_u = \sum_{i \in hits_u} \frac{1}{2^{\frac{rank(i)-1}{\alpha}}} \tag{2.9}$$

The final score requires the normalization of the rank score, thus it is necessary to compute the maximum ranking score using equation 2.10, which returns the maximum achievable scores if all the items selected by the user were assigned to the lowest possible rankings, i.e. ranked according to the bijective function $idx(i)$, which assigns values from 1 to $\mid hits_u \mid$ to the items in $hits_u$.

$$RankScore_u^{max} = \sum_{i \in hits_u} \frac{1}{2^{\frac{idx(i)-1}{\alpha}}} \tag{2.10}$$

Finally, the normalized rank score is given by equation 2.11.

$$RankScore_u^{'} = \frac{RankScore_u}{RankScore_u^{max}} \tag{2.11}$$

Here we have presented only the most relevant metrics for evaluating the performance of recommender systems, several alternative metrics can be found in the literature (Herlocker et al., 2004; Jannach et al., 2011).

2.4.3 Context-Aware Recommendation

Most of the approaches used in current recommendation systems are mainly focused in estimating how relevant is an item to an user, ignoring any contextual information that could be used to improve the recommendation process. However, context-based recommendation systems are emerging, taking context into account when providing recommendations to the user (Adomavicius and Tuzhilin, 2011).

According to Adomavicius and Tuzhilin (Adomavicius and Tuzhilin, 2011), context can be integrated in the recommendation process, where ratings can be modeled taking into account contextual information, besides being based on users and items. Within this scope, context can be defined from a *representational* point of view (Dourish, 2004), using a predefined set of observable attributes that do not change significantly over time, or from an *interactional* prespective, which models context through a short-term memory interactional approach (Anand and Mobasher, 2007). The contextual information can be obtained either *explicitly* or *implictly*, or can be *inferred*. It is obtained explicitly when it is gathered directly from people or relevant information sources. Sometimes it is not possible, or adequate, to obtain contextual information directly and it must be implicitly captured from the data or the environment. Finally, contextual information can be inferred using statistical or data mining techniques (Witten and Frank, 2005).

The contextual information can be used by a recommender system via *context-driven querying and search* or *contextual preference elicitation and estimation* (Adomavicius and Tuzhilin, 2011). The former approach is usually based on using contextual information to query a repository of resources and select the most relevant resources. The later approach is a recent trend and involves modeling and learning the user preferences, by monitoring the interactions of the users or obtaining their feedback. Depending on the way context is used, systems using contextual information follow a *contextual pre-filtering*, *contextual post-filtering*, or *contextual modeling* paradigm. The contextual pre-filtering paradigm uses the contextual information to restrict the data set to a reduced set of relevant items, which can then be recommended using a traditional recommendation approach. The contextual

post-filtering paradigm uses contextual information to filter or rank a list of recommendations obtained using a traditional recommendation approach. The contextual modeling paradigm uses the contextual information directly in the rating estimation process, and can be further divided in *heuristic* and *model* based approaches. Some recommender systems combine multiple approaches, either to overcome some of the drawbacks of individual approaches or adapt the recommendation process to different scenarios.

2.5 Software Development

The *software development* (McCarthy and McCarthy, 2006) activity is devoted to the development of a software product. Although it may include all the activities that result in a software product, from its conception to its realization, here we are focused on the process of writing and maintaining the source code. This process is usually conducted by developers in an Integrated Development Environment (IDE)[7], which is a software application aimed to help developers during software development activities. These applications usually comprise a source code editor, build tools and a debugger, although some of them may also contain other tools, such as a compiler or an interpreter. Among the most popular IDEs are Eclipse[8], Netbeans[9], IntelliJ IDEA[10] and Microsoft Visual Studio[11]. The Eclipse IDE (see figure 2.3) is one of the most used for the Java[12] programming language (Goth, 2005), although it can also be used for developing applications in other programming languages, by means of various plug-ins. Due to its wide dissemination, open source nature and extensibility features, Eclipse has been addressed in several academic studies (Robillard et al., 2004; Ko et al., 2006; Sillito et al., 2006; Murphy et al., 2006; LaToza et al., 2007; Sillito et al., 2008; Starke et al., 2009) and has served as a basis for the development of several exploratory tools to help developers in their activities (Janzen and De Volder, 2003; Robillard and Weigand-Warr, 2005; Singer et al., 2005; Holmes and Murphy, 2005; Mandelin et al., 2005; Poshyvanyk et al., 2006a; Sahavechaphan and Claypool, 2006; Kersten and Murphy, 2006; Warr and Robillard, 2007; Hummel et al., 2008; Zhong et al., 2009; Ratanotayanon et al., 2010; Piorkowski et al., 2012).

Over the recent years, a number of studies have been carried out to understand the activities of the developer, especially how they comprehend software (von Mayrhauser and Vans, 1995; LaToza et al., 2007), investigate the source code (Robillard et al., 2004; Ko et al., 2006), use the IDE (Murphy et al., 2006), deal with interruptions (González and Mark, 2004), and what are their information needs (Sillito et al., 2006; Ko et al., 2007; Sillito et al., 2008). Here we will present two of these studies, one by Ko et al. (Ko et al., 2006) and another by Sillito et al. (Sillito et al., 2008), which are mainly focused on understanding how developers seek, collect and relate relevant information, as well as what kind of questions they have and how these questions are answered by existing tools.

Having the objective of improving the developer's effectiveness on maintenance tasks, Ko et al. (Ko et al., 2006) performed an exploratory study to answer four questions:

1. How do developers decide what is relevant?
2. What type of relevant information do they seek?
3. How do they keep track of relevant information?

[7] `http://en.wikipedia.org/wiki/Integrated_development_environment` (August 2012)
[8] `http://www.eclipse.org/` (August 2012)
[9] `http://netbeans.org/` (August 2012)
[10] `http://www.jetbrains.com/idea/` (August 2012)
[11] `http://www.microsoft.com/visualstudio/` (August 2012)
[12] `http://www.oracle.com/us/technologies/java/` (August 2012)

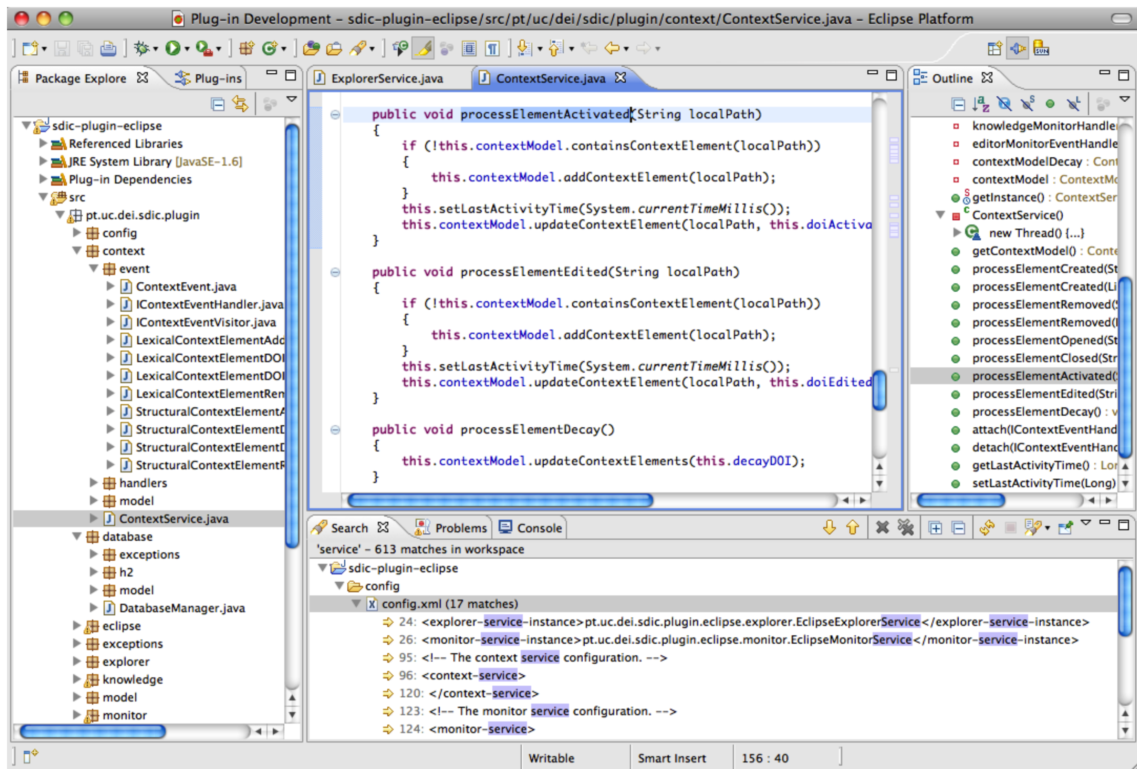

Figure 2.3: A screenshot of the Eclipse IDE.

4. How do their task contexts differ on the same task?

Their study included 10 developers using the Eclipse IDE to perform five maintenance tasks, three debugging tasks and two enhancement tasks, in a system unknown to them and within a 70 minute period. During the experiment, the developers were artificially interrupted with the aim of mimicking the real interruptions that occur in a real scenario. They have recorded the work of the developers using a screen capture application and then analyzed the videos.

With respect to the labor division, the developers spent a little more of a fifth of their time handling interruptions. The non-interrupted time was spent performing different activities, a quarter of it was spent performing *textual searches* and *navigating* the source code, a fifth of it *reading* code and another fifth *editing* code. They have observed that the developers followed a higher-level sequence of actions, which included choosing the task to work on, searching for task-relevant information, understanding the relationships between information and then dealing with the necessary code. When developers needed to search for task-relevant information, most of them began with a textual search for what they thought to be a task-relevant identifier in the code. The debugging tasks began with searches for symptoms and surface features of the program's failure. But, an average of 88% of these searches led to the investigation of irrelevant code, mainly because the identifiers in the code do not fully represented the code's purpose. The enhancement tasks began with searches for extension points in the source code, which could be used as examples for implementing the new features.

When developers needed to determine the relevance of source code or related information, they would go through different levels of information and several cues. For instance, they would began with the *package explorer*, looking to the name of the file and its icon and deciding then if they would *open* the file or simply *expand* its node in the tree. The

developers who expanded the nodes would *inspect* the names of methods and fields, looking for something interesting. While the developers who open a file generally *skim* the source code, looking for identifiers or comments that might explain the intent of the file. During these investigations, developers have found different types of relevant information, including source code that would be edited, duplicated or used to understand the behavior of other relevant code.

After analyzing a piece of code, developers also explored its incoming and outgoing dependencies, usually by following static relationships. About 58% of them were *direct* dependencies, which could be found through direct static relations, the other 42% were *indirect* dependencies, which were usually related with elements indirectly related by two or more static relations. Most of the exploring of direct dependencies was performed using less sophisticated tools, such as the *find and replace* dialog. The authors point out that the complexity and/or cost of using more powerful tools, such as *Java Search* or *Open Declaration*, could be the reason for this behaviour. For exploring indirect dependencies, the developers needed to *scroll* the source code, use the *package explorer* and go through *file tabs*.

Regarding the task contexts, developers used different kinds of strategies to keep track of relevant source code and information. They used the *package explorer* and *file tabs* to keep track of files, the *scroll bars* and *text carret* to mark relevant segments of code, *bookmarks* to mark lines of code and also the *undo stack* to access early versions of their code. Some of them even used *paper notes* to remember relevant information. The authors conclude that although the interfaces provided were helpful, they were not sufficient for handling the developer needs. From their observations, the authors concluded that the impact of interruptions in the work of developers were significant when developers did not *externalize the task state* to the environment when the interruption was acknowledged, and when they could not *recall the state* after returning from interruption.

Based on their findings, the authors propose a model for program understanding, which describes this task as a process of *searching*, *relating* and *collecting* relevant information. As central factors for the success of a developer, they point out that the environment must provide clear and representative cues for the developer to judge the relevance of information, and provide a reliable way of collecting the information the developer considers relevant. With respect to tools, they have concluded that the support for navigation between relevant source code and information could be improved, as 35% of their non-interrupted time was spent on *searching* and *navigating* the source code. Especially, the tools should be able to provide better relevance cues, for instance to better understand the purpose and intention of the source code. Also, these cues should be provided across different layers, for instance to avoid the need to inspect the information in full to decide what is, or is not, relevant. They also suggest that tools should make easier the navigation through the dependencies of related source code. Finally, they propose that the existing tools should help developers collect and compare information side by side.

Sillito et al. (Sillito et al., 2008) performed two qualitative studies in order to understand what a programmer needs to know about a code base when performing change tasks, how does s/he search that information and how much existing tools help s/he on getting that information. The first study was conducted in a laboratory setting, with 9 participants working with source code that was new to them. The participants were asked to work in pairs and perform a change task in a time period of 45 minutes. During the sessions, the discussion was audio recorded, the action in the screen was captured in video and a log of several IDE events was made. In the end, the experimenter briefly interviewed the participants about the experience. The second study was conducted on a large technology company, with 16 programmers working on source code for which they had responsibility. The participants were asked to perform tasks they have chosen, in a

time period of 30 minutes. The sessions were audio recorded and an interview was made by the experimenter by the end of each session.

The authors used a graph of entities, and relationships between those entities, representing the code base, to categorize the questions asked by developers during the study sessions. A set of 44 different types of questions were categorized in four main categories:

1. Finding focus points;
2. Expanding focus points;
3. Understanding a sub-graph;
4. Questions about groups of sub-graphs.

The "finding focus points" category refers to questions about finding starting points in code, points that are relevant to a task or entities corresponding to specific aspects of the system. The "expanding focus points" category includes questions about expanding an entity believed to be important for a task, which is achieved by expanding its incoming and outgoing relationships. The "understanding a sub-graph" category implies understanding the concepts behind multiple relationships and entities, which includes understanding a certain behavior or data/control flow. The "questions over groups of sub-graphs" category contains questions focused on comparing groups of sub-graphs in order to understand their relationships and how a change to those structures will impact the rest of the system. As expected, the questions in the first three categories appeared more frequently in the first study, while those in the fourth category were more frequent in the second study.

Concerning the tool support for answering the 44 types of questions identified, they have analyzed which tools were available and the level of support provided. The questions pertaining to the first category were typically answered by performing searches for hypothetical identifiers or textual references. Although answering questions in this category was considered well supported, there were some difficulties on formulating queries and dealing with the amount of information returned. Answering questions in the second category generally includes gathering information about different types of relationships between source code entities. This task is typically supported by static analysis, debugging, overview and data-flow tools. Despite some forms of indirection and the volume of information presented by these tools, there is a good tools support in this case. The tools available for answering questions in the third category were limited, although some questions could be depicted using program slicing, debugging and code browsing tools. Finally, the questions included in the fourth category were also difficult to answer, despite the use of diff, code cloning detection and testing tools.

They conclude by pointing out that questions in the first two categories could be easily answered with current tools. But questions in the last two categories, which implied the combination of information about different points of the source code, have a poor support in todays tools. The authors suggest that a more comprehensive support is needed in three related areas: support for more refined and precise questions, support for maintaining context, and support for piecing information together. The support for more refined and precise questions is needed because tools typically limit the possibility of defining the scope on which to operate, forcing programmers to generalize their questions and end up retrieving too many irrelevant results. Because most of the tools provide isolated information, answering a question may involve gathering information from different sources with results that are largely undifferentiated and unconnected. Improved support for maintain context would help programmers answer higher level questions. The same way, support for piecing information together would help programmers assemble all the information needed to answer their questions.

The two studies presented before provide an analysis of what are the information needs of developers, how they collect and relate the needed information and how existing tools support their efforts. In the following sections, we focus on the role that search and recommendation tools may have in supporting the activities of the developers.

2.5.1 Search

As demonstrated by several studies (Ko et al., 2006; Sillito et al., 2008; Starke et al., 2009), source code search is an essential activity for a developer working on an IDE. This activity is supported by a range of search tools that are available in most of the today's IDEs, but little is known about their usage patterns and effectiveness. With the aim of analysing the search activities of developers, Starke et al. (Starke et al., 2009) have performed a study to understand how developers conduct their searches, explore the search results and decide which results are relevant, during a change task. The study involved ten developers working in two change tasks in a large software system that was unknown to them. The developers worked in 30 minute sessions, using all the features provided by the Eclipse IDE, including eight major kinds of search tools available in Eclipse, namely:

- *Open Type.* Search for classes or interfaces based on a partial name or pattern.
- *File.* Search for text within all of the files in the workspace.
- *Find in File.* Search for a piece of text within a specified file.
- *Java.* Search for declarations, references and occurrences of source code elements (packages, types, methods and fields).
- *References.* Search for all references to a specified source code element, or elements, matching a keyword.
- *Implementors.* Search for all classes that implement a specified interface, or interfaces, matching a keyword.
- *Declaration.* Search for all declarations of a specified source code element, or elements, matching a keyword.
- *Occurrences in File.* Search for all occurrences of a specified source code element in the current file.

They have collected both quantitative and qualitative data about 96 complete search episodes, that were conducted using the search tools provided in Eclipse. From the analysis of this data, they have depicted five key observations:

1. The participants formed hypothesis about the problem based on their experience, which was very important to begin exploring the source code and guide the remaining work in the task;
2. They have performed searches based on their initial hypothesis, placing themselves in the situation of the original developers and creating search queries based on naming conventions, synonyms and different ways to express similar ideas;
3. They had only an idea of what to search for and of what would be relevant, so their searches were often generic, leading to a high number of search results and a lack of confidence about the relevance of these results to the task;

4. Instead of investigating the search results in detail, the developers generally skimmed through the results, trying to find evidence of their relevance in the information provided by the search interface, such as the packages or names;

5. The developers tended to open a small number of search results, and most of the times the source code of the opened elements was only skimmed, often starting new searches when they did not find any clue of relevance.

Based on their observations, the authors propose several enhancements that could help mitigate the problems faced by developers when searching for relevant source code elements. Because developers often used the information available in the search interface, such as structural information and element names, to evaluate the relevance of search results, more contextual information could be provided to help determine the relevance of a search result. The problem of dealing with many search results at once could be partially solved with an improved ranking mechanism, which should also take into account the context of the developer. Finally, several enhancements could be applied to the search interfaces, in order to improve the search experience and guide the exploration of the search results.

2.5.2 Recommendation

A Recommender System for Software Engineering (RSSE) has been defined by Robbilard et al. (Robillard et al., 2010) as *"a software application that provides information items estimated to be valuable for a software engineering task in a given context"*. The increasing dimension and complexity of software development projects are fostering the development of such systems, which have been applied to very different tasks in software development, such as software reuse, expertise location, code comprehension, guided software changes, debugging, etc.

As noted by Happel and Maalej (Happel and Maalej, 2008), the today's work of a software developer demands using diverse technologies and complex frameworks and coping with a large amount of continually changing information, as well as dealing with strict deadlines, limited resources and changing priorities. These authors point out that being a knowledge and automation intensive domain, software development is a good target for recommendation systems that help developers answering their questions. They argue that pro-active recommendations should address both information seekers and information providers, instead of focusing only on the information seekers perspective of recommending "what similar developers like". They present a review of state of the art approaches and discuss some of the limitations, and related challenges, that affect such approaches. Among the limitations identified, they distinguish:

- The limitation to either recommend methods to use next or artifacts related to the current situation;
- The dependence on centralized and static corpus;
- A limited description of context including single properties such as the current class a user is working in;
- No pro-active triggering of information push;
- Inflexible architectures that do not allow for extensions.

From the limitations identified, they discuss some challenges for future systems. Regarding architectures, they suggest the use of decentralized approaches, instead of a client/server style, for encouraging information sharing and avoid performance issues. With

respect to knowledge representation, it is proposed that more flexible knowledge representation structures, such as those based on the Semantic Web (Berners-Lee et al., 2001) technologies, would improve the information integration and sharing, as well as setting the ground for making the system behaviour more transparent. Concerning pro-activeness, they state that a better description of the user context would allow for more focused and pertinent recommendations. Also, they consider that recommendations should go beyond explicit knowledge, for instance, by trying to identify problem solving patterns that would be relevant for developers.

Based on their analysis of existing tools, they also propose a "landscape" for software development recommendation systems, depicting improvements that could be achieved by addressing the stage of the recommendation process and the types of knowledge recommended. With respect to the former, they suggest that intelligent push of information for a given context of the user is desirable. Also, they claim that a recommendation system should encourage the users to share certain information that could be useful within their teams. Concerning the types of knowledge that should be recommended, they make a distinction between development and collaboration information. As development information, they highlight source code, relevant artifacts, quality measures and appropriate tools, as the best candidates for recommendation. While people, awareness and information about status and priorities, are pointed out as collaborative information that could be provided to developers by appropriate recommendation systems. Finally, they identify context awareness and the concept of "inverse search" as the basic building blocks of future recommendation systems. Because the sharing of context information raises privacy questions, an "inverse search" mechanism would allow the existence of a private information model, with only some parts of this model being anonymously shared with the community.

According to Robillard et al. (Robillard et al., 2010), the design of a RSSE can be analyzed from three different perspectives: nature of context, recommendation engine and output mode (see table 2.1). The *recommendation context* represents the input of a RSSE, and it can be explicit, implicit or a hybrid of these. The context information can be explicitly provided by the developer, for instance by indicating a set of elements of interest, or it can be implicitly gathered, for instance through the analysis of the interactions of the developer in the IDE. An hybrid approach may combine explicit information provided by the developer with additional information that is gathered implicitly. The *recommendation engine* may use additional types of data to make its recommendations, including source code, system changes, artifacts, interaction history, etc. These recommendations can be ranked according to their predicted relevance for the developer, which usually depends on the task and the developer together. The *output mode* of a RSSE depends on the way recommendations are provided to the developer. When the developer explicitly requests for recommendations, the RSSE is said to work on pull mode. When recommendations are automatically delivered to the developer, the RSSE works on push mode. The delivery of recommendations can be processed in batch mode, when the recommendations are presented in a separate zone of the IDE, or inline mode, if the recommendations are presented as annotations over the desired artifacts.

Some of the features of a RSSE are considered cross-dimensional, including the user feedback and explanations. The *user feedback* can be taken into account to improve the ranking of recommendations, either locally, through explicit adjustments or an adaptive mechanism, or globally. The *explanations* provided by the system allow the developer to understand the origin of recommendations, which may help building their trust in the system, but may also carry some pitfalls, such as information overload.

Table 2.1: The RSSE design dimensions, adapted from (Robillard et al., 2010).

<table>
<tr><th>Nature of Context</th><th>Recommendation Engine</th><th>Output Mode</th></tr>
<tr><td rowspan="3">Input
(explicit | implicit | hybrid)</td><td>Data
(source | changes | bug reports | mailing lists | interaction history | peer's actions)</td><td>Mode
(push | pull)</td></tr>
<tr><td>Ranking
(yes | no)</td><td>Presentation
(batch | inline)</td></tr>
<tr><td colspan="2">Explanations
(from none to detailed)</td></tr>
<tr><td colspan="3">User Feedback
(none | locally adjustable | locally adaptive | globally adaptive)</td></tr>
</table>

2.6 Summary

This chapter provided an introduction to the theoretical background that supports the research developed in this thesis. This theoretic introduction will be complemented with a detailed description of a more practical work, presented in chapter 6. Here we make a bridge between the theoretical work described in the previous sections with the approach we have followed in our work, which will be described in chapter 3.

We started with an introduction to the concept of ontology, as a knowledge representation structure, its classification approaches, building process and applications. We are especially interested in a less formal definition of ontology, whereby an ontology can be used to represent entities and the relations that exist between them. This type of ontology was used in our knowledge base (see section 3.1), to represent the source code stored in the workspace of the developer.

Then, context and its ambiguous nature was described, along with the approaches used for modeling contextual information. Although context has been addressed in a generic way, we have introduced the theoretic concepts behind the definition of a context model (see section 3.2), which is one of the cornerstones of our approach. We are interested in context as the set of circumstances and facts that surround the center of interest, providing additional information and increasing understanding. We apply this definition to software development, more specifically to the work of the developer in an IDE.

The IR field, which provides the framework for any textual search process, was presented and described in relation to modeling approaches, text processing operations, query formulation, evaluation and context-awareness. The VSM and the TF-IDF term weighting approach, which were described in the scope of the IR modelling approaches, are in the basis of our context-based search process (see section 3.3), where they are used to retrieve and rank the source code elements based on a query provided by the developer.

The field of recommender systems, which are in the basis of any recommendation process, was also presented, providing the theoretical framework that are in the basis of our context-based recommendation process (see section 3.4).

Finally, we have introduced the field of software development, with a special focus on how developers search and collect relevant information for their needs. We have described a set of studies focusing on the developer needs, especially with respect to search and recommendation, that help framing our approach within the field of software development.

Chapter 3

Approach

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein

This chapter describes the mechanisms that comprise the basis of our approach to context-based retrieval in software development. We start by discussing the work environment of a software developer from a broader perspective. The work of a developer covers several dimensions that are not limited to the work developed on an Integrated Development Environment (IDE). We have defined a context model of the developer, as illustrated in figure 3.1, where these dimensions are represented in a layered model, including a *personal layer*, a *project layer*, an *organization layer* and a *community layer*. The contextual information of the developer crosses these four layers, focusing on different aspects according to the targeted dimension. At each layer, the context of the developer can be used to improve the retrieval of information that is relevant for her/his work, for instance through *search*, *recommendation* or *browsing*.

The *personal layer* represents the work a developer has at hands at any point in time, which can be defined as a set of tasks. In order to accomplish these tasks, the developer has to deal with various kinds of resources at the same time, such as source code files, specification documents, bug reports, etc. These resources may be dispersed through different places and systems, although being connected by a set of explicit and implicit relations that exist between them. At this level the context model represents the resources that are important for the tasks the developer is working on. For instance, when the developer is working on a specific task, there are a set of resources that are more relevant for that task than others, which can be highlighted to the developer.

The *project layer* focuses on the project, or projects, in which the developer is involved. A software development project is an aggregation of a team, a set of resources, and a combination of explicit and implicit knowledge that keeps the project running. The team is responsible for accomplishing tasks, which end up consuming and producing resources. The relations that exist between people and resources are the glue that makes everything work. The project layer represents the people and resources, as well as their relations, of the software development projects where the developer is included. For instance, when fixing a specific bug, it is important to know what other bugs might be related, which files are likely to be affected, and which other developers working on the project may be of help to solve the bug.

The *organization layer* takes into account the organization to which the developer belongs. Similarly to a project, an organization is made up of people, resources and their relations, but in a much more complex network. While in a project the people and

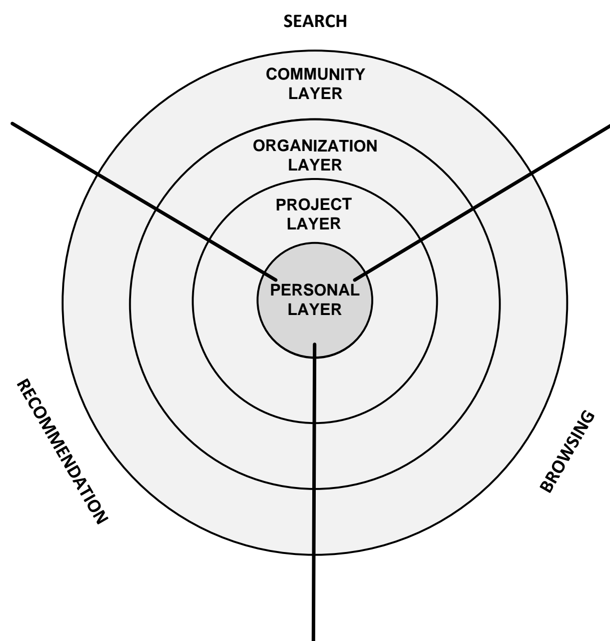

Figure 3.1: The layered context model of the developer, crossing the different dimensions that comprise her/his work environment.

resources are necessarily connected due to the requisites of their work, in an organization these projects easily become separate islands. The knowledge and competences developed in each project may be of interest in other projects and valuable synergies can be created when this information is available. The organization layer represents the organizational context that surrounds a developer. For instance, when a developer needs to apply a specific technology with which no one in the project team is familiar with, there may be other colleagues, or even resources, in the same organization which can help.

The *community layer* takes into account the knowledge domain, or domains, in which the developer works. This layer goes beyond the project and organization levels, including a set of knowledge sources that stand out of these spheres. Nowadays, a typical developer uses the Internet to search information and to keep informed of the advances in the technologies s/he works with. These actions are based on services and communities, such as source code repositories, foruns, mailing lists, blogs, etc. These knowledge sources cannot be detached from the developer context and are integrated in the community layer of our context model. For instance, due to the dynamic nature of the software development field, the developer must be able to gather knowledge from sources that go beyond the limits of the organization, either to follow the technological evolution or to search for help whenever needed.

The research work described here is devoted to the personal layer of the context model presented above. More specifically, we have focused on how the contextual information of the developer could be used to improve the search and recommendation of source code in the IDE. The following sections describe the different components that comprise our approach (see figure 3.2). We start by describing the *knowledge base*, which provides a formal representation of the source code stored in the workspace of the developer. The source code is represented from a structural and a lexical perspectives. We describe how these perspectives are formalized using a *structural ontology* and a *lexical ontology*, how these ontologies are built and how the source code is indexed for later retrieval. Then, we describe the *context model* that is used to represent the source code elements that

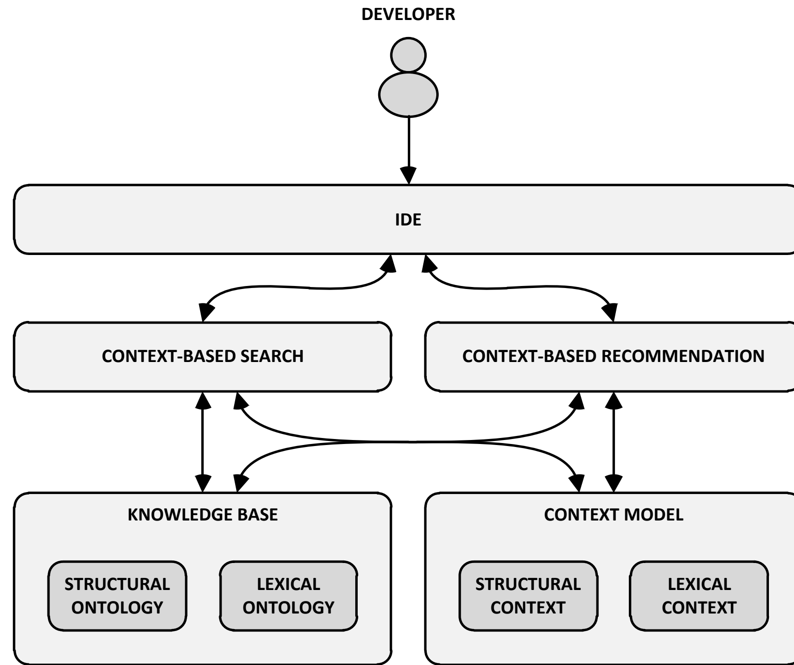

Figure 3.2: The conceptual architecture of our approach to context-based retrieval in software development.

are more relevant for the developer in a specific moment. As in the knowledge base, the context model also comprises a structural and a lexical perspectives, which are presented as the *structural context* and the *lexical context*. The context model is built based on the interactions of the developer with the source code, and adapts itself as the focus of attention of the developer changes. The knowledge base and the context model are the foundations for the *context-based search and recommendation* approaches that are described next. These mechanisms make use of these two models to improve the retrieval and ranking of source code elements, when the developer searches for them or when the system predicts they are relevant for the developer. Finally, we describe a *learning* mechanism that adapts the ranking of source code elements according to the components that are more relevant for each developer.

3.1 Knowledge Base

The *knowledge base* represents the source code structure that is stored in the workspace of the developer. This knowledge base is unique for each developer, being built from the source code files with which the developer is working, and maintained as these files are changed. The source code structure is represented from a structural and a lexical perspectives, which are formalized using ontologies. The structural perspective deals with the source code artifacts and the structural relations that exist between them, while the lexical perspective deals with the terms used to reference these artifacts and how they are associated. In this section, we provide a detailed description of the ontologies used to represent the knowledge base structure. Then, the process used to build and maintain the knowledge base is presented. Finally, we describe how the knowledge is indexed for later retrieval.

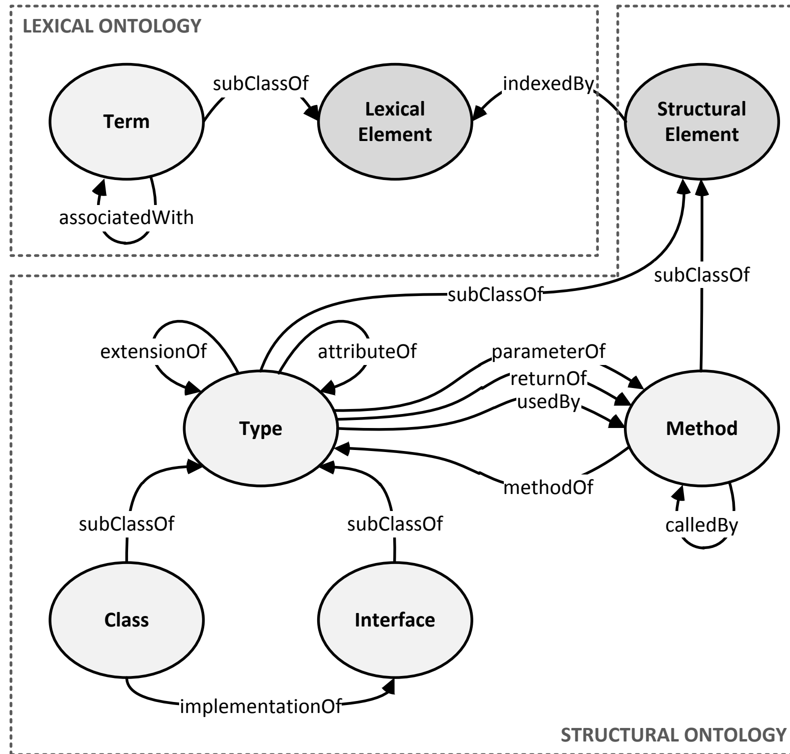

Figure 3.3: The structural and lexical ontologies model.

3.1.1 Ontologies

The source code structure represented in the knowledge base is formalized using ontologies, which are used as an explicit representation of entities and their relations. The structural and lexical perspectives of the source code will be described as two separate ontologies, for the sake of comprehension, but they can be represented in an unique model, as presented in figure 3.3. Next, we describe the structural and lexical ontologies in more detail.

Structural Ontology

The *structural ontology* represents a set of source code elements typically found in object-oriented programming languages, as well as a subset of their most relevant relations (see figure 3.3). The model was inspired by the source code structure used in the Java programming language (Gosling et al., 2005), but can be easily adapted to other object-oriented programming languages. The main source code elements we have represented in our structural ontology are *classes* and *methods*, which are the building blocks of an object-oriented programming language. Additionally, we have also included *interfaces*, because they are commonly used as a way to enforce behaviour in classes. These source code elements are represented in the class hierarchy of the structural ontology by classes with the same name, namely Class, Interface and Method, defined as sub-classes of an abstract class named Structural Element. The Class and Interface classes are also defined as sub-classes of the abstract class named Type, which aggregates some of the characteristics shared by the two classes. The concrete implementations of classes, interfaces and methods, in the source code, are represented as instances of the corresponding classes. The source code elements represented in the structural ontology are connected by a set of relations that are used in object-oriented programming to express *inheritance* (extensionOf and implementationOf), *composition* (attributeOf and methodOf) and *behavior* (parameterOf, returnOf,

calledBy and usedBy), which can be described as follows:

- extensionOf relates a class with another class that is extended by the former.
(e.g. `MySQLDatabaseManager` → extensionOf → `DatabaseManager`)
- implementationOf relates a class with an interface that is implemented by that class.
(e.g. `DatabaseManager` → implementationOf → `IDatabaseManager`)
- attributeOf relates a class with another class that has the former as an attribute.
(e.g. `Connection` → attributeOf → `DatabaseManager`)
- methodOf relates a method with a class that declares that method.
(e.g. `addProduct(Product)` → methodOf → `DatabaseManager`)
- parameterOf relates a class with a method that receives an object of that class as a parameter.
(e.g. `Product` → parameterOf → `addProduct(Product)`)
- returnOf relates a class with a method that returns an object of that class.
(e.g. `Product` → returnOf → `getProduct()`)
- calledBy relates a method with another method that calls the former.
(e.g. `getProduct()` → calledBy → `updateProduct(Product)`)
- usedBy relates a class with a method that uses an object of that class.
(e.g. `Product` → usedBy → `updateProduct(Product)`)

Lexical Ontology

The *lexical ontology* represents the terms used to reference the source code elements, including a set of relations that are used to express how terms relate to each other and with the source code elements (see figure 3.3). The concrete *terms* used to compose the name, also known as *identifier*, of a source code element are represented as instances of the Term class, which is defined as a sub-class of the abstract class named Lexical Element. The terms used to reference a source code element become *indexed* by that element using the indexedBy relation. Each term instance can be associated to more than one source code element, because it may be used to compose the name of different elements. The number of times a term is used to index a source code element represents its frequency in the knowledge base and is also stored. When two terms are used together to compose the name of a source code element, we create an associatedWith relation between them. This relation is used to represent the *proximity* between the terms, the same way co-occurrence is interpreted as an indicator of semantic proximity in linguistics (Harris, 1954). We interpret co-occurrence of two terms in identifiers as an indication of some kind of relation between that terms. The terms that co-occur more often have a stronger relation than those that are rarely used together. Therefore, the number of times the two terms co-occur in the names of different structural elements is stored and used as the weight of the relation between these terms.

3.1.2 Building

The knowledge based is built from the source code files that are stored in the workspace of the developer. As illustrated in figure 3.4, these files are parsed in order to extract the source code elements that they represent. Then, the source code elements extracted go through a reference disambiguation process and are stored in the structural ontology.

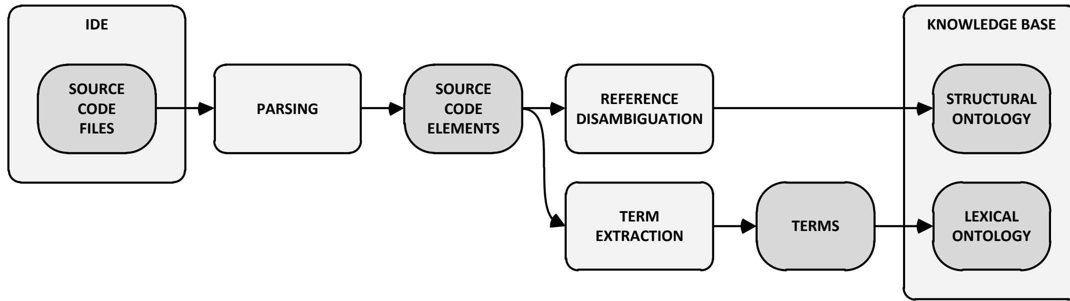

Figure 3.4: Abstract representation of the process used to build the knowledge base.

The terms used to reference the source code elements are also extracted and stored in the lexical ontology. We will now describe each one of these steps in more detail.

The source code structure stored in the workspace of the developer is typically organized as a set of projects containing several source code files. The knowledge base is automatically built through the analysis of these source code files, and maintained as they change over time. The source code elements, as well as their relations are extracted from source code files through *parsing*. A parsed source code file results in an Abstract Syntax Tree (AST), representing the abstract syntactic structure of the source code in the form of a tree. A simplified AST, generated for the source code of listing 3.1, is presented in figure 3.5. The source code of listing 3.1 represents a Compilation Unit, which comprises a Package Declaration, an Import Declaration and a Type Declaration. The Type Declaration accounts for the declaration of the `DatabaseManager` class, which contains three Method Declarations, namely `addProduct`, `getProduct` and `updateProduct`. The body of method `updateProduct` contains a Variable Declaration, named `currentProduct`, which is initialized through a Method Invocation of method `getProduct`. The AST is used to extract information about classes, interfaces and methods, and how they relate with each other. The parsing does not require a source code file to be compilable, it only needs it to be syntactically correct. This way, we are able to process all the source code files that are well constructed, even if they are not compilable. After parsing the source code of listing 3.1 and generating its corresponding AST, represented in figure 3.5, the structural ontology would be populated as illustrated in figure 3.6.

Although we are able to identify the source code elements that are locally declared in each file, these elements are usually related with elements that are implemented in different files. The references to these external elements may not include the information necessary to unambiguously identify that element within the entire source code structure. This information is available at compile type, because the compiler has access to the entire source code structure and is able to create the proper bindings between this references and their correspondent elements. Because we parse each file individually, we do not have a view of the entire source code and have to deal with this problem in a process called *reference disambiguation*. This process requires that the source code elements and their relations, extracted during the parsing process, are added to the knowledge base in two distinct phases. At first, all the source code elements are added to knowledge based at once. Then, the relations between the source code elements are processed. A relation between elements that have been unambiguously identified are directly added, while those that contain ambiguous references must be disambiguated. The disambiguation process tries matching the ambiguous element in the same package of the referencing element, in a direct import or in package import, by searching the element in the elements that were already added to the knowledge base. Following the example source code of listing 3.1, the type `Product` is referenced several times, but in none of them the full reference, which

Listing 3.1: Example of part of the Java source code implementing a class.

```
1  package example.database;
2
3  import example.database.model.*;
4
5  public class DatabaseManager implements IDatabaseManager
6  {
7        [...]
8
9        public void addProduct(Product product)
10       {
11             [...]
12       }
13
14       public Product getProduct(String id)
15       {
16             [...]
17       }
18
19       public boolean updateProduct(Product product)
20       {
21             [...]
22
23             Product currentProduct = getProduct(id);
24
25             [...]
26       }
27
28       [...]
29 }
```

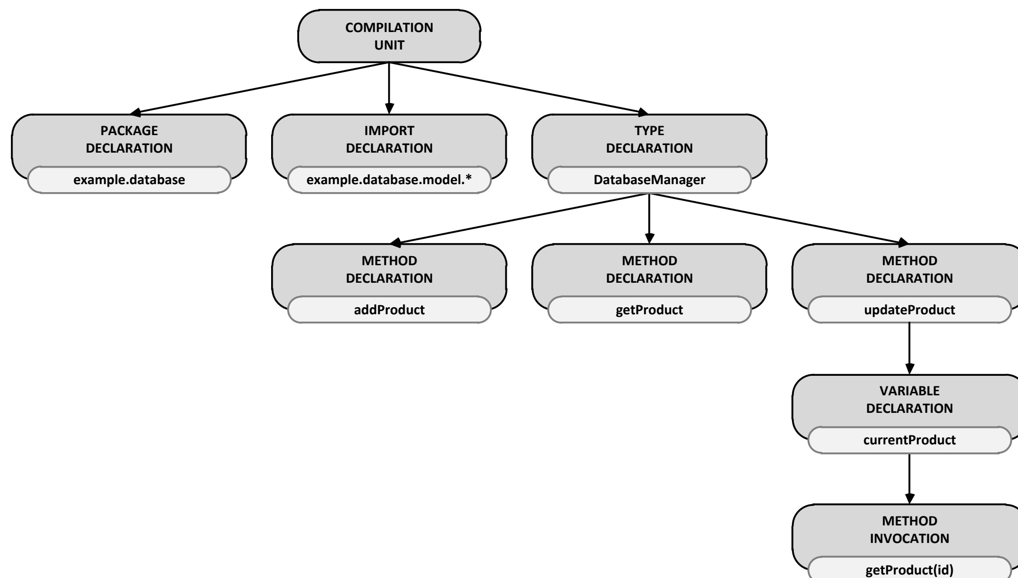

Figure 3.5: Example of a simplified AST generated for the source code of listing 3.1.

would be `example.database.model.Product`, is provided. This way, when extracting the structural relations, the type `Product` must go through the reference disambiguation process, until its reference is found through the import declared in line 3 of listing 3.1.

The terms associated to each source code element are extracted from its name, or *identifier*, using a *term extraction* process based on a set of rules. The identifiers of such elements usually comprise more than one term, to better express the semantics associated to each element. The naming conventions used in software development regulate how the identifiers of classes, interfaces, methods or variables should be created and how the different terms that comprise these identifiers should be joined. The naming convention used in the Java programming language is *CamelCase*[1], also known as medial capitals,

[1] `http://en.wikipedia.org/wiki/CamelCase` (August 2012)

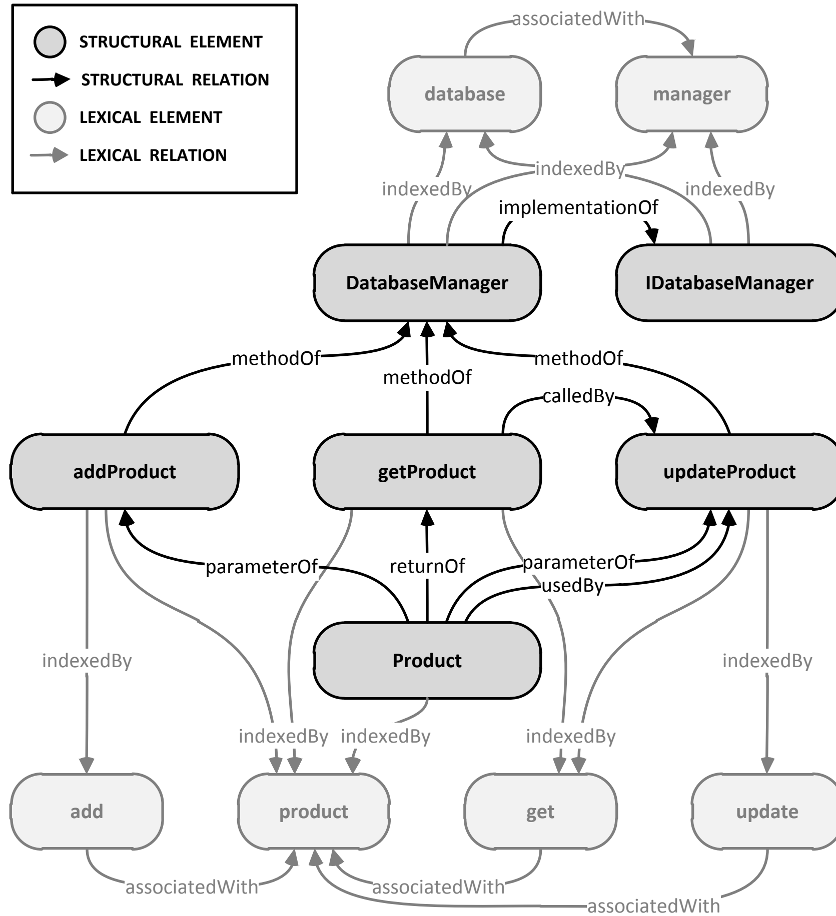

Figure 3.6: Example of how the source code of listing 3.1 is represented in the structural and lexical ontologies.

which defines that the terms comprising an identifier should be joined without spaces, with the initial letter of each term in capitals. The convention also suggests that whole words, instead of abbreviations, should be used, unless the abbreviation is more widely used that the long form.

The term extraction process we use is based on this naming convention. The naming convention is not enforced, developers are free to define an identifier using other conventions or their own rules. But these conventions are followed by the majority of the developers, because they assure coherence and improve the readability of the source code. This way, we use a set of regular expressions to split identifiers into their individual terms, based on the rules expressed in the naming convention. Additional rules provide that acronyms are correctly extracted, identifiers with separator characters can be broken and terms with only one character are discarded. The set of rules used can be summarized as follows:

1. Split the identifiers into individual terms that are joined by the special character '_' (e.g. `result_set` → `result` | `set`).

2. If the terms, extracted using the previous rule, contain isolated capital letters, split the terms using these capital letters (e.g. `DatabaseManager` → `database` | `manager`).

3. If the terms, extracted using the previous rule, contain sequences of capital letters, split the terms using these sequences (e.g. `SQLQuery` → `sql` | `query`).

4. If an extracted term contains only one character, discard this term.

Although this strategy will not work correctly with all the identifiers, the widespread use of the naming convention assures it will work most of the times. In figure 3.6, one can observe how the terms extracted from each source code element become represented in the lexical ontology.

Because the source code is constantly being modified, when the developer changes the source code files in the workspace, the knowledge base must be updated accordingly. The workspace of the developer is constantly monitored for changes. When a source code file is added, removed or updated, the knowledge base is updated as follows:

- *Add.* The source code file being added is parsed and the elements/relations extracted are added to the knowledge base.
- *Remove.* All the elements/relations associated with the file being remoded are removed from the knowledge base.
- *Update.* The elements/relations associated with the previous version of the file being updated are first removed, then the file is parsed again and the extracted elements/relations are added to the knowledge base.

3.1.3 Indexing

The source code elements represented in the knowledge base are indexed for later retrieval. The indexing of these elements follows the Vector Space Model (VSM) (Salton et al., 1975), an algebraic Information Retrieval (IR) model that is among the most used in several IR systems (see section 2.3.1). In our approach, a *document* represents a source code element, such as a class, an interface or a method, stored in the knowledge base. Thus, the *collection of documents* represents all the source code elements that exist in the workspace of the developer. The source code elements, or documents, are represented as a vector of weights associated to the *terms* extracted from their identifiers. The terms in these documents are weighted using Term Frequency/Inverse Document Frequency (TF-IDF) (Salton and Buckley, 1988), which computes the weight of a term based on its frequency in the document and the inverse of its frequency among the collection of documents (see section 2.3.1). Although we here describe the theoretic aspects related to the indexing of the source code elements, we did not implement the IR approach used, instead we have used an existing search engine, as described in section 4.1.

The contents of the documents in our document collection, which represent source code elements, comprises lines of source code, thus these documents can not be processed as common textual documents. The only textual parts that can be found in source code are usually located in identifiers, comments and string literals. In fact, most of the IR approaches applied to source code use one, or more, of these textual fragments for indexing source code elements (see chapter 6). We opted to use only the terms contained in the identifiers to maintain simplicity and objectivity. We believe that these terms, being used to name the source code element, are those that best describe the semantics associated to that element. The problems of using the source code comments are that they require additional processing to identify the most relevant terms and are often missing in source code.

As described before, the identifiers commonly comprise more than one term. This way, each source code element is indexed by the terms extracted from its identifier, using the same approach described in section 3.1.2. Additionally, we also index the source code elements by all combinations of the individual terms, in the same order they appear in the

Table 3.1: Example of the terms used to index the source code elements of listing 3.1.

Element	Terms
IDatabaseManager	idatabasemanager, idatabase, databasemanager, i, database, manager
DatabaseManager	databasemanager, database, manager
addProduct	addproduct, add, product
getProduct	getproduct, get, product
updateProduct	updateproduct, update, product
Product	product

identifier. This was done to deal with situations when the terms are incorrectly extracted, or when the combination of terms, as expressed in the identifier, have some meaning. The terms used to index the source code elements represented in listing 3.1 are presented in table 3.1.

3.2 Context Model

The *context model* we have defined aims to represent the *focus of attention* of the developer in each moment and is based on the source code elements that are more relevant to her/his work in that moment. The model is built from the interactions of the developer with the source code elements and evolves over time, as the focus of attention of the developer changes. Similarly to the knowledge base, the context model comprises a structural and a lexical dimensions. The process of building the context model from the developer interactions is illustrated in figure 3.7 and will be described in more detail on the following sections. Then, we present an approach to adapt the context model according to the changes in the focus of attention of the developer.

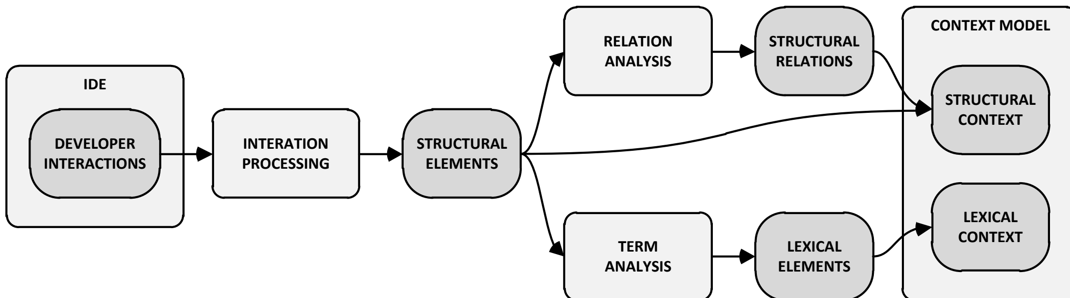

Figure 3.7: Abstract representation of the process used to build the context model.

3.2.1 Structural Context

The *structural context* focuses on the *structural elements* and *structural relations* that are more relevant for the developer in a specific moment. The relevance of these elements and relations is derived from the interactions of the developer with the source code and is represented as an *interest* value. The structural context was inspired by the work of Kersten and Murphy (Kersten and Murphy, 2006), which have used a similar model to represent the context associated to a task (see section 6.1). Next, we describe the structural context in more detail, including the structural elements and structural relations.

Table 3.2: List of captured interactions, their description and interest variation.

Interaction	Description	Variation
Open	When the developer accesses an element for the first time.	+0.4
Activate	When the developer accesses an element that was previously accessed.	+0.2
Edit	When the developer edits an element.	+0.1
Close	When the developer closes an element.	−0.4

Structural Elements

The *structural elements* account for the source code elements with which the developer is interacting, including classes, interfaces and methods represented in the knowledge base (see section 3.1.1). Associated to each element is an *interest* value that is derived from the interactions of the developer with that element. The list of structural elements and their interest is continuously updated, so that the interest of an element can be directly consulted at any time, without the need to perform any extra calculation.

The interest of an element changes according to the different interactions that affect that element. The impact of each interaction has been defined based on our experience and some empirical tests. The variation applied to the interest of an element upon a certain interaction reflects how that interaction contributes to increase or decrease the relevance of that element in the context of the developer. The list of interactions that are taken into account, their description and the variation applied to the interest of the affected element are presented in table 3.2. When an element is *opened*, or accessed for the first time, it is added to the structural context and its interest is increased. When the element is *activated* or is *edited*, its interest is increased at a lower rate, because these interactions tend to be more repetitive. The more the developer accesses or edits an element, the more relevant it becomes in the context model. When an artifact is *closed*, its relevancy to the developer decreases, but it is not immediately removed from the context model, as it may still be relevant in the current context of the developer.

As time passes, the interest of the elements must be decayed, so that the relevance of an element in the context of the developer decreases if it is not used over time. The *decay* may be processed in different ways, for instance, as applied in previous works (Kersten and Murphy, 2006; Parnin and Gorg, 2006), the interest of an element can be decreased proportionally to the number of interactions of the developer over time. We opted to use an approach based on time, so that the interest of the elements is decreased as time passes. This way, the interest of an element is not affected by the interactions of the developer with other elements, it is based only on the interactions that affected that element and the time passed since its last interaction. The decay should be applied to each element individually but, for reasons of simplicity, we apply the decay to all the elements at a fixed time interval. In the current implementation, the decay is processed every five minutes and applies a variation of -0.1 to the interest of every structural element in the context model. These parameters were defined according to our own experience and observations during the implementation of the approach. In order to prevent loosing context when the developer is distracted, or away for some reason, the decay is executed only if the developer has been active in the IDE since the last decay. When the interest of an element reaches zero, that element is finally removed from the structural context. Consider SE the set of structural elements represented in the knowledge base KB, as defined in equation 3.1.

$$SE = \{se_1, se_2, \ldots, se_n \mid se_i \in KB\} \tag{3.1}$$

Table 3.3: Example of how a set of consecutive interactions affects the interest of a structural element.

Interaction	**Variation**	**Interest**	**Normalized Interest**
Open	+0.4	0.4	≈ 0.33
Activate	+0.2	0.6	≈ 0.45
Edit	+0.1	0.7	≈ 0.50
Edit	+0.1	0.8	≈ 0.55
Decay	−0.1	0.7	≈ 0.50
Activate	+0.2	0.9	≈ 0.59
Edit	+0.1	1.0	≈ 0.63
Close	−0.4	0.6	≈ 0.45

Let $I(se)$ be the interest associated to an element se, then the structural elements represented in the context model would be the subset of structural elements in the knowledge base with a positive interest ($CSE \subseteq SE$), as defined in equation 3.2.

$$CSE = \{se_1, se_2, \ldots, se_n \mid se_i \in KB \wedge I(se_i) > 0\} \quad (3.2)$$

Consider E an event that affects an element se, including interactions and decays, and E^{Δ} the variation applied to the interest of the element se affected by the event E. Then, the interest of an element is computed using equation 3.3, where $I'(se)$ denotes the interest associated to the structural element se, before normalization, and E_i^{Δ} represents the ith event of the n events that affected element se.

$$I'(se) = \sum_{i=1}^{n} E_i^{\Delta} \quad (3.3)$$

As the developer interacts with the source code elements, their interest grows without restriction. This could lead to a situation in which an element could obtain a disproportionate interest in relation to the remaining elements. This would make it difficult to compare the relative relevance of such an element in relation to the relevance of other elements. Also, we may argue that beyond some threshold, the interest of an element can be considered so high that the real value of its interest is irrelevant. Taking this into account, the final interest of an element is always normalized to the interval $[0, 1]$. The normalized interest $I(se)$ is computed using an inverted exponential function, as shown in equation 3.4, assuring that the interest of an element has an exponential growth, becoming very close to 1 for values over 5. In table 3.3, we provide an example of how a set of consecutive interactions affects the interest of a structural element.

$$I(se) = 1 - \left(\frac{1}{e^{I'(se)}}\right) \quad (3.4)$$

As illustrated in figure 3.8, the structural elements `Product`, `DatabaseManager`, `addProduct`, `getProduct` and `updateProduct` were added to the structural context because they were manipulated by the developer at some point in time. The relevance of these elements to the developer is represented by their interest values, which evolves according to the different interactions of the developer with that elements over time.

Structural Relations

The *structural relations* account for the relevance of the relations that exist between the source code elements that are being manipulated by the developer. These relations correspond to the structural relations that are defined in the structural ontology (see section 3.1.1). The relevance of the structural relations can be used to measure the relevance of other source code elements, that may not being used by the developer but are structurally related with the elements that are in the structural context.

The structural relations are not directly affected by the interactions of the developer, therefore, their relevance is derived from the structural elements that exist in the structural context. When two, or more, structural elements are bound by one of these relations, that relation is added to the structural context. Associated with each relation is an interest value that represents the relevance of that relation in the context of the developer. The interest of a relation is computed as an average of the interest of all structural elements that are bound by that relation. This way, the relevance of a structural relation reflects the relevance of the structural elements that brought it to the structural context. The structural relations and their interest are updated whenever the structural context changes. When no more structural elements are bound by a relation, it is removed from the structural context. Consider SR the set of relations defined in the structural ontology SO, as defined in equation 3.5.

$$SR = \{sr_1, sr_2, \ldots, sr_n \mid sr_i \in SO\} \tag{3.5}$$

Let $I(sr)$ be the interest associated to relation sr, then the structural relations represented in the context model would be the subset of structural relations of the structural ontology with a positive interest ($CSR \subseteq SR$), as defined in equation 3.6.

$$CSR = \{sr_1, sr_2, \ldots, sr_n \mid sr_i \in SO \land I(sr_i) > 0\} \tag{3.6}$$

The interest of a structural relation is computed using equation 3.7, where se_i^{sr} represents the ith structural element of the n structural elements bound by relation sr.

$$I(sr) = \frac{\sum_{i=1}^{n} I(se_i^{sr})}{n} \tag{3.7}$$

The structural relations represented in figure 3.8, namely `parameterOf`, `usedBy`, `returnOf`, `methodOf` and `calledBy`, were added to the structural context because there are structural elements bound by these relations. These relations were illustrated in figure 3.6, section 3.1.2, where an example of the knowledge base is provided. The interest associated with these relations represents the average interest of the elements that are bound by them.

3.2.2 Lexical Context

The *lexical context* focuses on the terms that are more relevant in the context of the developer. The terms are extracted from the names of the source code elements that are manipulated by the developer, using the same approach used to extract the terms represented in the knowledge base (see section 3.1.2). Similarly to the elements and relations in the structural context, the relevance of each term is given by an interest value. The interest of a term is computed as an average of the interest of the structural elements from which the term was extracted. The more relevant is a structural element, the more relevant become the terms used to reference that element. The relevance of these terms can be used to identify source code elements that are lexically related with the source

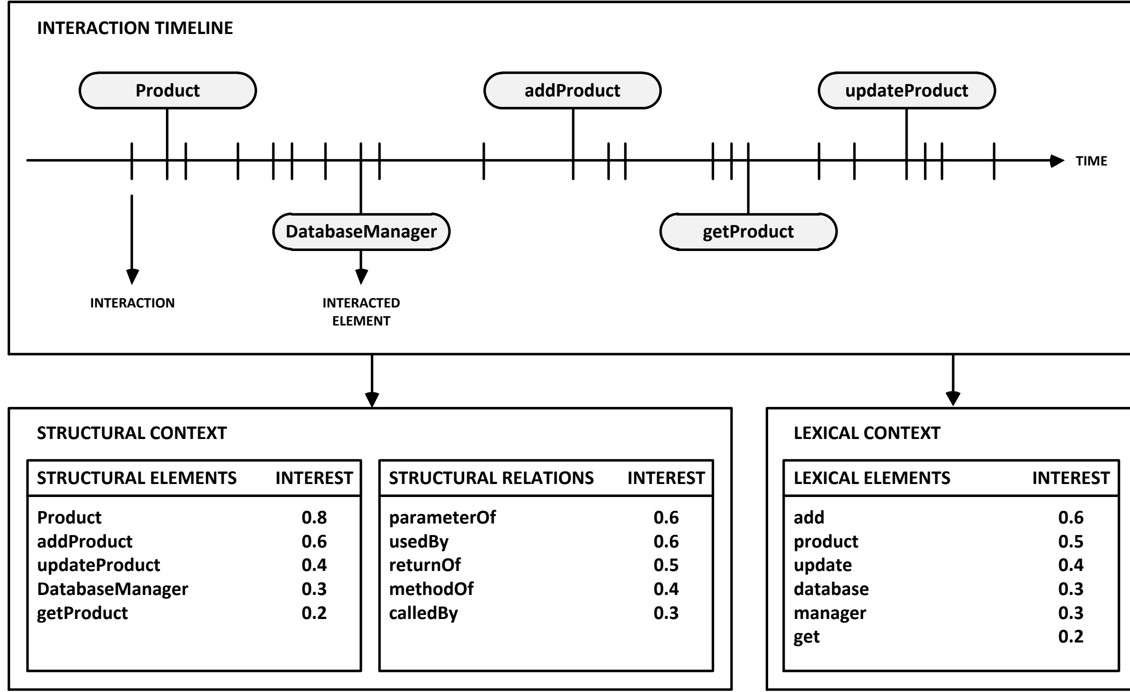

Figure 3.8: Example of how the context model is derived from a set of interactions in the interaction timeline.

code elements that are currently relevant for the developer. When the structural elements change, the lexical context is updated accordingly. Consider LE the set of all lexical elements contained in the knowledge base KB, as defined in equation 3.8.

$$LE = \{le_1, le_2, \ldots, le_n \mid le_i \in KB\} \tag{3.8}$$

Let $I(le)$ be the interest associated to the lexical element le, then the lexical elements represented in the context model would be the subset of lexical elements in the knowledge base with a positive interest ($CLE \subseteq LE$), as defined in equation 3.9.

$$CLE = \{le_1, le_2, \ldots, le_n \mid le_i \in KB \wedge I(le_i) > 0\} \tag{3.9}$$

The interest of a lexical element is computed using equation 3.10, where se_i^{le} represents the ith structural element of the n structural elements from which the lexical element le was extracted.

$$I(le) = \frac{\sum_{i=1}^{n} I(se_i^{le})}{n} \tag{3.10}$$

The terms in the lexical context represented in figure 3.8 are `add`, `product`, `update`, `database`, `manager` and `get`. These terms were extracted from the names of the elements in the structural context. Their interest represents the average interest of the structural elements referenced by them.

3.2.3 Context Transitions

As the focus of attention of the developer changes, the notion of what is relevant to her/his work also changes and the context model must be adapted accordingly. The context model

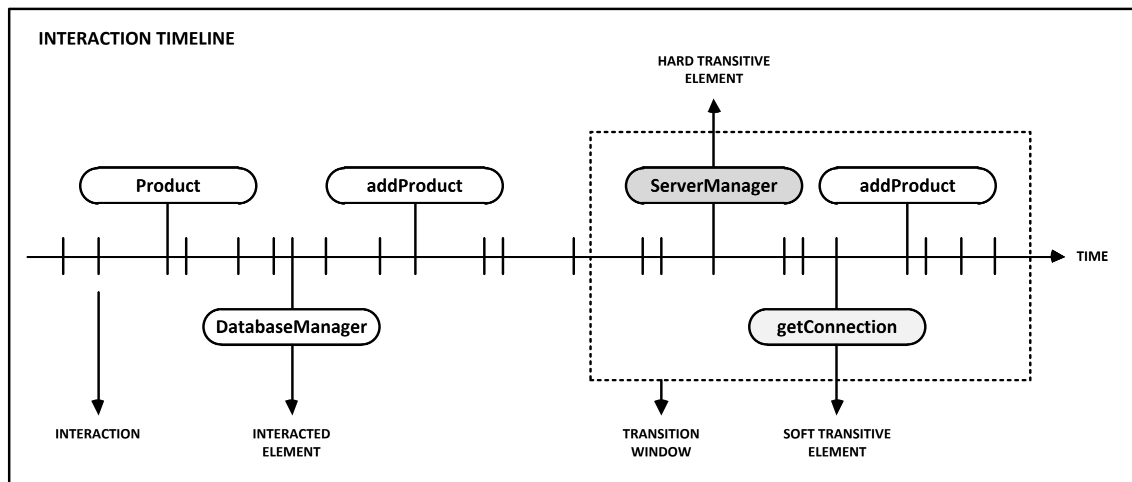

Figure 3.9: Example of how the context transition window is applied to the interaction timeline.

described before was designed to represent the focus of attention of the developer in each moment, but does not provide, by itself, the mechanisms needed to adapt as the focus of attention changes, which sometimes happens very fast. Because the developer commonly addresses more than one task in a short period of time, or even at the same time (Kersten and Murphy, 2006), the focus of attention is dispersed through different parts of the source code structure. This means that, in fact, more than one context model exist in parallel, and they must be activated and deactivated as the focus of attention changes. This issue has been addressed with a mechanism that deals with *context transitions*, as the focus of attention of the developer changes. Therefore, we may have several context models, that are stored in a context model pool, and only one context model active in each moment. The system automatically detects the changes in the focus of attention of the developer and decides whether a new context should be created or an existing one should be activated.

The changes in the focus of attention of the developer are detected based on how the source code elements added to the context model are related with those that are already in the context model. Our assumption is that when the attention of the developer shifts to a different part of the source code structure, it is expected to see, in a short period of time, a reasonable number of interactions with source code elements that have no relation with those in the current context model. In such a situation, the system must adapt to the change that is occurring in the behaviour of the developer and make a transition to a new context, or an existing one. To detect these situations we have defined a mechanism based on a *transition window* and a set of *transitive elements*. The transition window is a fixed time window that represents the time span within which a certain number of elements, having no relation with the current context, will start a context transition. We call these elements as transitive elements, and they can be either hard or soft transitive.

The *hard transitive* elements are those that have no close relation with the elements in the current context, whereas the *soft transitive* elements are those that have some kind of relation with only hard transitive, or other soft transitive, elements. The distinction between soft and hard transitive elements was necessary due to the fact that after accessing an element that has no relation with the current context, the developer commonly accesses other elements that are related with this element. This happens, for instance, when the developer opens a class and then edits some of its methods or opens the interface implemented by that class. If we would rely only on the hard transitive elements, it

would be very difficult to detect a context transition, because we would never observe a considerable number of unrelated elements within the transition window. This way, we use the soft transitive elements to represent the elements that, although being related with other elements in the context model, are only related with elements that are already marked as transition elements. The transition window moves along the interaction timeline as time passes. The hard or soft transitive elements that reach the limit of the time window are no longer marked as transitive elements. As illustrated in figure 3.9, the transition window is located at the head of the interaction timeline. There are three elements that were accessed within the transition window, one is a hard transitive element, one is a soft transitive element and the other is an element that was already in the context model. The `ServerManager` class is considered a hard transitive element because it has no relation with any of the previously manipulated elements, while the `getConnection` element is considered a soft transitive element because it is a method of the `ServerManager` class but is not related with the remaining elements in the context model. The `addProduct` method is not considered a transitive element because it was manipulated before and was already in the context model.

Transition Detection

A context transition is detected when a considerable number of hard or soft transition elements is observed within the transition window, which is interpreted as a change in the focus of attention of the developer. A change in the focus of attention of the developer can be viewed from different perspectives, depending on the source code structure, the tasks that are being addressed and the developer. Consequently, the size of the transition window and the number of transitive elements necessary to trigger a context transition varies with the situation.

Nevertheless, during the implementation and fine tuning of our approach we found a set of parameters that have been used to detect context transitions. When the number of hard transitive elements reaches a threshold of 3, or the number of soft transitive elements reach a threshold of 6, within a context transition window of 3 minutes, a context transition is initiated. These parameters were defined based on our own experience and observation of different situations that, in our perspective, would represent a change in the focus of attention of the developer. They may work well in most of the situations, but unavoidably will be inaccurate in others. When a new structural element is being added to the context model, we use algorithm 1 to detect if a context transition must be processed or not. This algorithm references a set of functions, which are described as follows:

- UpdateContextTransitionWindow(). This function updates the transition elements, contained in the context transition window, by verifying if the last access time of these elements remains within the 3 minute frame. The transition elements whose last access time is out of the 3 minutes frame are no longer marked as transitive elements.
- ProcessContextTransition(). This function processes the transition to a new or an existing context, depending on the situation.
- IsHardTransitive(structuralElement). This function verifies if the structural element being processed is a hard transitive element.
- IsSoftTransitive(structuralElement). This function verifies if the structural element being processed is a soft transitive element.
- AddContextElement(structuralElement). This function adds the structural element being processed to the active context model.

Algorithm 1: ProcessContextElement

Input: A *structuralElement* to be added to the context model.

```
1  begin
2      UpdateContextTransitionWindow()
3      if structuralElementCount > hardElementThreshold then
4          if IsHardTransitive(structuralElement) then
5              if hardElementCount = hardElementThreshold then
6                  ProcessContextTransition()
7              else
8                  AddContextElement(structuralElement)
9      else if structuralElementCount > softElementThreshold then
10         if IsSoftTransitive(structuralElement) then
11             if softElementCount = softElementThreshold then
12                 ProcessContextTransition()
13             else
14                 AddContextElement(structuralElement)
15     else
16         AddContextElement(structuralElement)
```

Transition Processing

When a context transition is detected, the system must remove the transitive elements from the current context, deactivate it and decide if a new context should be created or an existing one should be activated. To activate an existing context, one must assure that the developer is changing the focus of attention to a part of the source code structure that originated the existing context. The system decides if an existing context should be activated by comparing its elements with the transitive elements, those that were used to detect the context transition in the first place.

For an existing context to be activated it has to contain all the transitive elements. A context model that contains all the transitive elements assures that the new focus of attention of the developer will be properly represented. Also, such a context model increases the odds that the remaining elements in that context model are also relevant, because they are very likely to be related with the transitive elements. In case that the activated context model does not correctly represent the new focus of attention of the developer, the context transition process will be there to detect this situation and process a new context transition, if necessary. When there are no other context models, or when the condition for activating an existing context model is not satisfied, a new context is created. The transitive elements are added to this new context and then it is activated.

3.3 Context-Based Search

The *context-based search* process that we have defined allows the developer to search for source code elements, such as classes, interfaces and methods, stored in the workspace. As illustrated in figure 3.10, the search results are retrieved using an IR based approach, which collects a set of source code elements that match a given query. These search results are then ranked according to their relevance to the query, but also taking into account their proximity to the context of the developer. We base our approach on the assumption that the source code elements the developer is looking for are likely to be related with

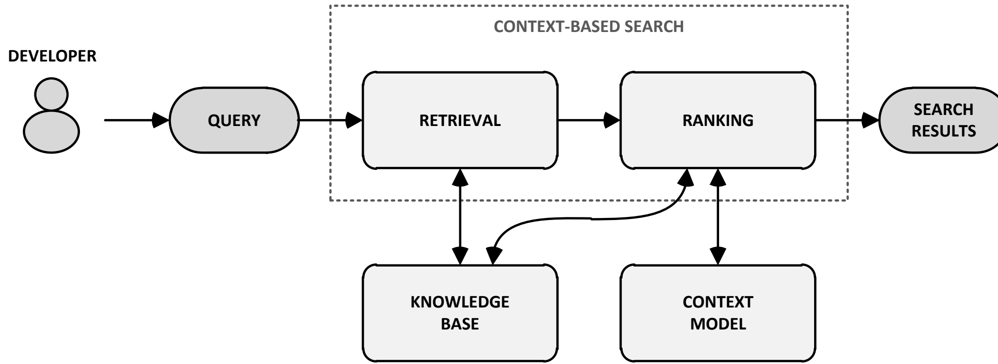

Figure 3.10: Abstract representation of the context-based search process.

the source code elements that are relevant in the current context. Next, we describe the retrieval process in more detail and explain how different components are combined to compute the relevance of a search result.

3.3.1 Retrieval

The retrieval of source code elements stored in the workspace of the developer is processed using an IR approach based on the VSM model (see section 2.3.1). As described in section 3.1.3, a document represents a source code element, such as a class, an interface or a method, stored in the knowledge base. Thus, the collection of documents represents all the source code elements that exist in the workspace of the developer. The source code elements, or documents, are represented as a vector of weights associated to the terms extracted from their identifiers. The retrieval process starts with a query provided by the developer. Similarly to the documents in the collection, the query is processed and transformed into a vector of weights associated to each one of the terms comprising the query. The documents indexed by one (or more) terms that match the terms in the query are retrieved. The retrieved documents are then ranked according to different components, which are described in the following section.

3.3.2 Ranking

The retrieved source code elements are ranked according to their relevance to the query and the context model of the developer, including the structural and lexical contexts. The relevance of a search result in relation to these components is given by a retrieval score (sr), a structural score (ss) and a lexical score (sl). Finally, the contribution of these components to the final score (sf) of the search result is given by a set of weights (w_r, w_s and w_l). The final score of the search result is computed as a weighted sum of the three scores, as shown in equation 3.11.

$$sf = (w_r \times sr) + (w_s \times ss) + (w_l \times sl) \tag{3.11}$$

$$w_r + w_s + w_l = 1 \tag{3.12}$$

The sum of the three weights is always one (see equation 3.12) and the value of each score is always normalized in the interval $[0, 1]$. Therefore, the final score of a search result is always within the interval $[0, 1]$. Next, we describe in detail how the retrieval, structural and lexical scores are computed.

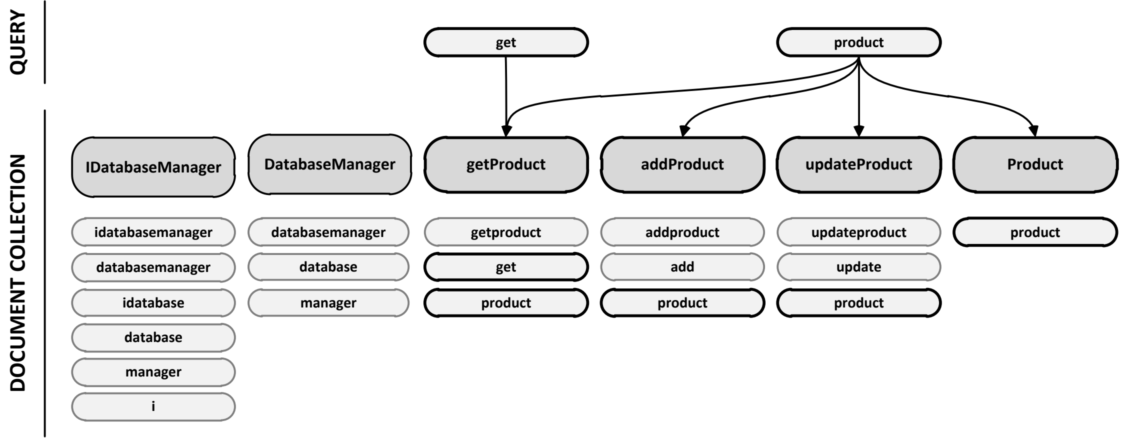

Figure 3.11: Example of the source code elements retrieved for a given query.

Retrieval Score

The *retrieval score* represents the relevance of the search result in relation to the query provided by the developer. As described before, both the source code elements, or documents, and the query are represented by a vector of term weights. The terms in documents and queries are weighted using TF-IDF (see section 2.3.1), which computes the weight of a term based on its frequency in the document and the inverse of its frequency among the collection of documents. The weight w of a term t in document d is given by equation 3.13, where f_{td} is the frequency of the term in the document, D is the number of documents in the document collection, and df_t is the frequency of the term in the document collection.

$$w(t, d) = f_{td} \times log\left(\frac{D}{df_t}\right) \tag{3.13}$$

The relevance of a document in relation to the query is represented as a correlation between the vectors of term weights associated to both the document and the query. This correlation is computed using the cosine similarity measure (see section 2.3.1), which represents the cosine of the angle between the two vectors. The cosine similarity between a query and a document is given by equation 3.14, which computes the dot product between the query vector $\vec{q}$ and the document vector $\vec{d}$.

$$sim(\vec{q}, \vec{d}) = \frac{\vec{q}}{\mid \vec{q} \mid} \cdot \frac{\vec{d}}{\mid \vec{d} \mid} \tag{3.14}$$

Therefore, the relevance of a search result in relation to the query provided by the developer is always a real value, within the interval $[0, 1]$, representing the level of match between the terms in the query and the terms extracted from the source code element, as shown in equation 3.15.

$$sr = sim(\vec{q}, \vec{d}) \tag{3.15}$$

As illustrated in figure 3.11, a query containing the terms `get` and `product`, would retrieve the documents representing the source code elements `getProduct`, `addProduct`, `updateProduct` and `Product`. An example of how the retrieval score would be computed for this query is given in table 3.4. We have included the frequency (TF) and inverse document frequency (IDF) for each term in the document collection. Then, we have computed the term weights for the query (Q) and each one of the documents (D_1, ...,

Table 3.4: Example of the term frequencies, inverse document frequencies, weights and scores computed for the query illustrated in figure 3.11.

	TF								Weight						
Term	**Q**	**D_1**	**D_2**	**D_3**	**D_4**	**D_5**	**D_6**	**IDF**	**Q**	**D_1**	**D_2**	**D_3**	**D_4**	**D_5**	**D_6**
add	0	0	0	1	0	0	0	1.79	0	0	0	1.79	0	0	0
addproduct	0	0	0	1	0	0	0	1.79	0	0	0	1.79	0	0	0
database	0	1	1	0	0	0	0	1.10	0	1.10	1.10	0	0	0	0
databasemanager	0	1	1	0	0	0	0	1.10	0	1.10	1.10	0	0	0	0
get	1	0	0	0	1	0	0	1.79	1.79	0	0	0	1.79	0	0
getproduct	0	0	0	0	1	0	0	1.79	0	0	0	0	1.79	0	0
i	0	1	0	0	0	0	0	1.79	0	1.79	0	0	0	0	0
idatabase	0	1	0	0	0	0	0	1.79	0	1.79	0	0	0	0	0
idatabasemanager	0	1	0	0	0	0	0	1.79	0	1.79	0	0	0	0	0
manager	0	1	1	0	0	0	0	1.10	0	1.10	1.10	0	0	0	0
product	1	0	0	1	1	1	1	0.41	0.41	0	0	0.41	0.41	0.41	0.41
update	0	0	0	0	0	1	0	1.79	0	0	0	0	0	1.79	0
updateproduct	0	0	0	0	0	1	0	1.79	0	0	0	0	0	1.79	0
Similarity										**0**	**0**	**0.04**	**0.72**	**0.04**	**0.04**

D_6). Finally, the similarity obtained between each document and the query ($sim(\vec{q}, \vec{d})$) is shown in the bottom of the table. The number of documents retrieved is limited to the top 100 documents with higher retrieval score, so that the ranking of the retrieved documents can be computed within an acceptable time frame, which was defined around $1s$. Although we here describe the theoretic aspects related to the retrieval of the source code elements, we did not implement IR approach used, instead we have used an existing search engine, as described in section 4.1.

Structural Score

The *structural score* represents the relevance of a retrieved search result in relation to the structural context. We define this relevance as the structural proximity between the source code element that was retrieved and the elements in the structural context. The structural proximity between two structural elements is inversely proportional to the structural distance between them. The task of measuring the structural distance between two source code elements is reduced to the problem of finding the shortest path between the two elements, by taking the structural ontology as a directed graph (Harary et al., 1965), where vertices are represented by structural elements and the edges are represented by structural relations. The cost of a path is given by the sum of the cost of the relations that create the path. Instead of using a fixed cost for each relation, the cost of a relation is inversely proportional to the interest associated to that relation in the structural context. This way, we take into consideration the current relevance of the structural relations to the developer, assuring that the paths created with more relevant relations will have a lower cost. Theoretically, we could use the entire graph to find the shortest path between a retrieved element and each one of the elements in the context model, for instance using the Dijkstra algorithm (Dijkstra, 1959). But this would not be feasible within acceptable time constraints. We also believe that when two elements are separated by a reasonable number of relations, the distance between them become too high to be considered. This way, we only consider the top 15 elements with higher interest in the structural context and perform a search for the shortest paths with a maximum of 3 relations.

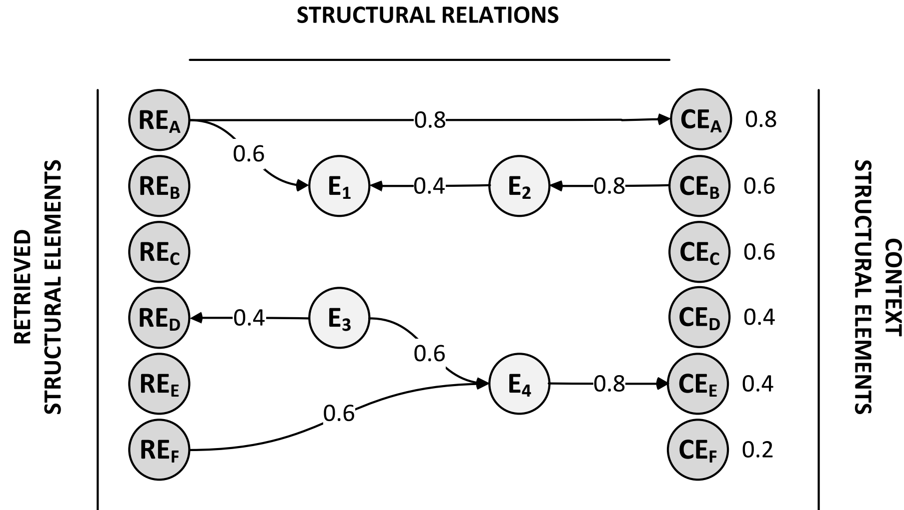

Figure 3.12: Example of the structural paths between a set of retrieved structural elements (RE) and the elements in the structural context (CE).

The structural distance between two structural elements is computed using equation 3.16, where sr_i is the ith relation of the n relations that create the shortest structural path between the elements se_a and se_b.

$$dist_s^{'}(se_a, se_b) = \sum_{i=1}^{n} 1 - I(sr_i) \tag{3.16}$$

The structural distance between the source code elements is normalized using equation 3.17, so that this distance is always a real number in the interval $[0, 1]$.

$$dist_s(se_a, se_b) = 1 - \left(\frac{1}{e^{dist_s^{'}(se_a, se_b)}} \right) \tag{3.17}$$

As shown in equation 3.18, the structural proximity between a source code element se and an element in the structural context ce is inversely proportional to the structural distance between the two elements, given by $dist_s(se, ce)$, and is proportional to the interest of the element in the structural context, given by $I(ce)$. This way, the lower the interest of the element ce to the developer, the lower is the proximity of an element in relation to ce.

$$prox_s(se, ce) = (1 - dist_s(se, ce)) \times I(ce) \tag{3.18}$$

The structural proximity between a source code element and the structural context is computed as an average of the structural proximity between that element and the elements in the structural context. Therefore, the structural score of a retrieved element r is given by equation 3.19, where ce_i represents the ith element within the list of n context elements.

$$ss(r) = \frac{\sum_{i=1}^{n} prox_s(r, ce_i)}{n} \tag{3.19}$$

An example of how the structural score is computed is illustrated in figure 3.12. The RE nodes represent the source code elements retrieved for a given query, the E nodes represent structural elements, and the CE nodes represent the elements in the structural context. We can find paths between these elements using the structural relations represented in the

structural ontology. For instance, between elements $\mathsf{RE_A}$ and $\mathsf{CE_B}$, there is a path through elements $\mathsf{E_1}$ and $\mathsf{E_2}$. The values associated to the CE nodes and the structural relations represent their interest in the structural context, respectively $I(ce)$ and $I(sr)$. We will now exemplify how the structural score of the retrieved element $\mathsf{RE_A}$ is computed. First, we need to find the structural distance between the element $\mathsf{RE_A}$ and the context elements $\mathsf{CE_A}$ and $\mathsf{CE_B}$, which are given by equation 3.16:

$$dist_s'(RE_A, CE_A) = (1 - 0.8) = 0.2$$
$$dist_s'(RE_A, CE_B) = (1 - 0.6) + (1 - 0.4) + (1 - 0.8) = 1.2$$

Then, the structural distance is normalized, using equation 3.17:

$$dist_s(RE_A, CE_A) = 1 - \left(\frac{1}{e^{0.2}}\right) \approx 0.1813$$
$$dist_s(RE_B, CE_B) = 1 - \left(\frac{1}{e^{1.2}}\right) \approx 0.6988$$

The structural proximity of the element $\mathsf{RE_A}$ to the context elements $\mathsf{CE_A}$ and $\mathsf{CE_B}$, formalized in equation 3.18, is computed using the the structural distance between these elements, as follows:

$$prox_s(RE_A, CE_A) \approx (1 - 0.1813) \times 0.8 \approx 0.6550$$
$$prox_s(RE_A, CE_B) \approx (1 - 0.6988) \times 0.6 \approx 0.1807$$

Finally, the structural score for the retrieved element $\mathsf{RE_A}$ is given by equation 3.19 and computed as follows:

$$ss(RE_A) \approx \frac{0.6550 + 0.1807}{2} \approx 0.14$$

Lexical Score

The *lexical score* represents the relevance of a retrieved search result in relation to the lexical context. We define this relevance as the lexical proximity between the terms associated with the source code element that was retrieved, which are extracted from its identifier, and the terms in the lexical context. This proximity is inversely proportional to the lexical distance between the terms. Similarly to the approach followed to compute the distance between two structural elements, the lexical distance between two terms is represented by the shortest path between them. Such paths can be found by taking the lexical ontology as a graph, where vertices are represented by terms and the edges are represented by the co-occurrence relations (associatedWith). We use the co-occurrence frequency of the two terms to measure the weight of their relation. We are assuming that the more frequent is the co-occurrence of two terms, the stronger is the relation between them. As shown in equation 3.20, the weight of a co-occurrence relation lr between terms t_a and t_b, is given by their co-occurrence frequency $cf(t_a, t_b)$, normalized by the maximum co-occurrence frequency in the knowledge base (cf_{max}).

$$w(lr_{t_a t_b}) = \frac{cf(t_a, t_b)}{cf_{max}} \tag{3.20}$$

This way, the cost of a relation is inversely proportional to the weight associated to that relation, so that more frequent relations connect terms with a lower cost. Accordingly, the cost of a path is given by the sum of the cost of the relations that create the path. By using the weight of the relations between terms to compute the cost of a path, we assure that the paths between terms that co-occur more frequently will have a lower cost. Again, we only consider the top 15 terms with higher interest in the lexical context and paths with a maximum of 3 relations. The lexical distance between two terms is computed using equation 3.21, where lr_i is the ith relation of the n relations that create the shortest lexical path between the terms t_a and t_b.

$$dist_l'(t_a, t_b) = \sum_{i=1}^{n} (1 - w(lr_i)) \tag{3.21}$$

The lexical distance between the two terms is normalized using equation 3.22, so that this distance is always a real number in the interval $[0, 1]$.

$$dist_l(t_a, t_b) = 1 - \left(\frac{1}{e^{dist_l'(t_a, t_b)}}\right) \tag{3.22}$$

As shown in equation 3.23, the lexical proximity between a term t and a term in the lexical context ct is inversely proportional to the lexical distance between the two terms, given by $dist_l(t, ct)$, and is proportional to the interest of the term in the lexical context, given by $I(ct)$. This way, the lower the interest of the term ct to the developer, the lower is the proximity of a term in relation to ct.

$$prox_l(t, ct) = (1 - dist_l(t, ct)) \times I(ct) \tag{3.23}$$

The lexical proximity between a term and the terms in the lexical context is computed as an average of the lexical proximity between that term and each one of the terms in the lexical context. The same way, the lexical proximity between a source code element and the terms in the lexical context is computed as an average of the lexical proximity between its terms and the terms in the lexical context. Therefore, the lexical score of a retrieved element r is given by equation 3.24, where t_i represents the ith term of the n terms associated to that element and ct_j represents the jth term of the m terms in the lexical context.

$$sl(r) = \frac{\sum_{i=1}^{n} \left(\frac{\sum_{j=1}^{m} prox_l(ti, ctj)}{m}\right)}{n} \tag{3.24}$$

An example of how the lexical score is computed is illustrated in figure 3.13. The RE nodes represent the source code elements retrieved for a given query, the T nodes represent terms, and the CT nodes represent the terms in the lexical context. The nodes $\mathsf{T_{A1}}$ and $\mathsf{T_{A2}}$ represent the terms extracted from the identifier of element $\mathsf{RE_A}$. The paths between the terms extracted from a retrieved element and the terms in the lexical context are also visible. For instance, between terms $\mathsf{T_{A2}}$ and $\mathsf{CT_C}$, there is a lexical path through terms $\mathsf{T_1}$ and $\mathsf{T_2}$. The values associated to the CT nodes represent their interest in the lexical context ($I(ct)$), while the values associated to the lexical relations represent the weight of that relation in the knowledge base ($w(lr)$). We will now exemplify how the lexical score of the retrieved element $\mathsf{RE_A}$ is computed. First, we need to find the lexical distance

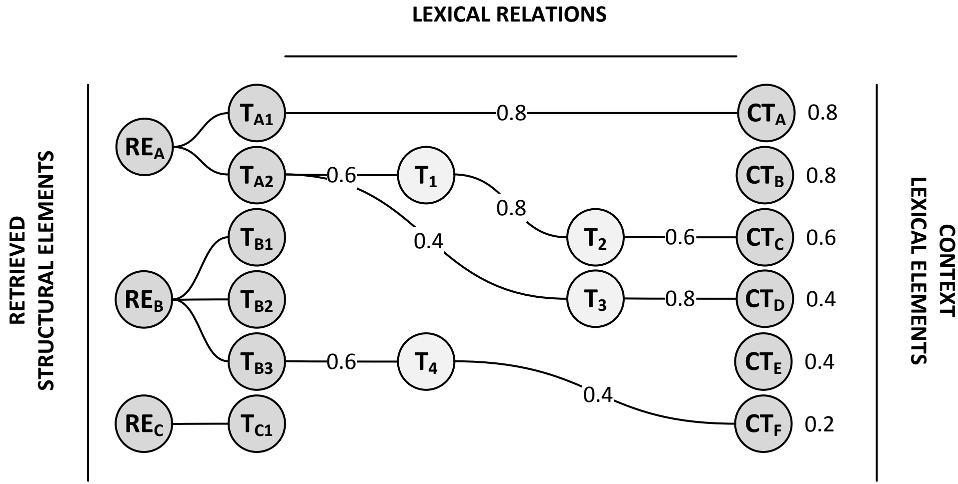

Figure 3.13: Example of the lexical paths between the terms extracted form a set of retrieved structural elements (RE) and the terms in the lexical context (CT).

between the term T_{A1} and the terms CT_A, and between the term T_{A2} and the terms CT_C and CT_D, which are given by equation 3.21:

$$dist_l'(T_{A1}, CT_A) = (1 - 0.8) = 0.2$$
$$dist_l'(T_{A2}, CT_C) = (1 - 0.6) + (1 - 0.8) + (1 - 0.6) = 1.0$$
$$dist_l'(T_{A2}, CT_D) = (1 - 0.4) + (1 - 0.8) = 0.8$$

Then, the lexical distance between these terms is normalized, using equation 3.22:

$$dist_l(T_{A1}, CT_A) = 1 - \left(\frac{1}{e^{0.2}}\right) \approx 0.1813$$
$$dist_l(T_{A2}, CT_C) = 1 - \left(\frac{1}{e^{1.0}}\right) \approx 0.6321$$
$$dist_l(T_{A2}, CT_D) = 1 - \left(\frac{1}{e^{0.8}}\right) \approx 0.5507$$

The lexical proximity of the terms extracted from element RE_A in relation to the terms in the lexical context is computed using equation 3.23, as follows:

$$prox_l(T_{A1}, CT_A) \approx (1 - 0.1813) \times 0.8 \approx 0.6550$$
$$prox_l(T_{A2}, CT_C) \approx (1 - 0.6321) \times 0.6 \approx 0.2207$$
$$prox_l(T_{A2}, CT_D) \approx (1 - 0.5507) \times 0.4 \approx 0.1797$$

Finally, the lexical score for the retrieved element RE_A is given by equation 3.24 and computed as follows:

$$sl(RE_A) \approx \frac{\left(\frac{0.6550}{6}\right) + \left(\frac{0.2207+0.1797}{6}\right)}{2} \approx 0.09$$

Additionally, we have also identified a set of terms (such as `get`, `set`, `add`, `to`, `is`, etc.) that appear very frequently in the name of the source code elements, especially methods.

These very frequent terms co-occur with a variety of other terms and end up connecting almost every term in a distance of a few relations, thus distorting our metric. This problem could be partially solved with a list of very frequent terms that would be ignored, but this would be very limiting, because the top frequent terms vary from workspace to workspace and may include terms that are specific to each workspace. This way, we ignore all the terms that would fall in the top 30% of all term occurrences. The paths that go through one of these terms are discarded. This percentage is based on our observation of a group of knowledge bases from different users, but needs to be further studied.

3.4 Context-Based Recommendation

The *context-based recommendation* process we have defined uses the context model of the developer to retrieve and rank potentially relevant source code elements, as illustrated in figure 3.14. This process is based on the assumption that most of the source code elements needed by the developer are likely to be structurally or lexically related with the elements that are being manipulated in that moment (see section 5.2.1). This way, we want to help developers reaching the desired source code elements more easily and quickly, decreasing the effort needed to search for that elements in the source code structure. The context model plays a central role in this process, providing the mechanism needed to identify and evaluate the relevance of the source code elements that are being manipulated. We use the structural elements represented in the context model, along with their relations, to retrieve recommendations of elements that are potentially relevant for the developer. These recommendations are then ranked taking into account different components, which represent both the retrieval process and the relevance to the context model.

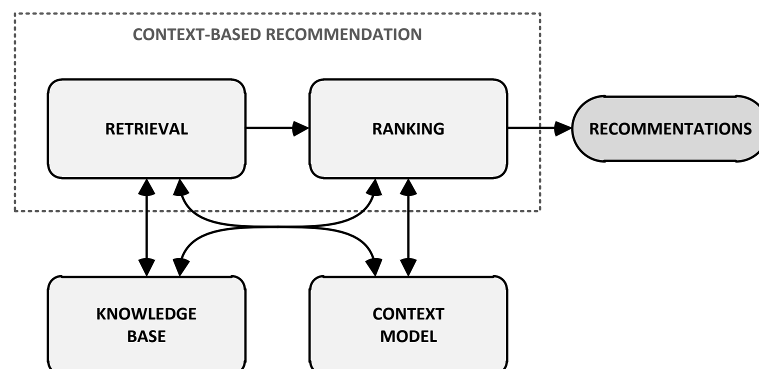

Figure 3.14: Abstract representation of the context-based recommendation process.

3.4.1 Retrieval

The recommendations are retrieved using the source code elements in the context model by combining two different methods, one based on the interest of these elements and other based on the time elapsed since these elements were last accessed. The first emphasizes the overall importance of the source code elements in the context of the developer, while the later privileges the elements that were recently accessed by the developer.

The *interest* based method makes use of the relevance of the source code elements that have been manipulated by the developer to identify other potentially relevant elements. The recommendations include the top N elements with higher interest in the context model and all the elements that are structurally related with them. We call N the query size of

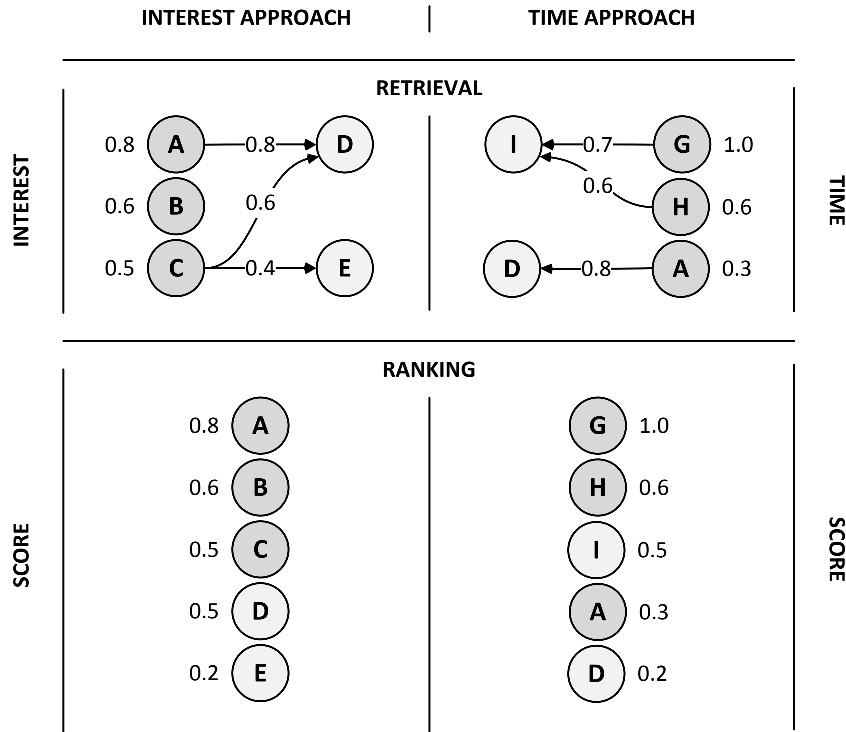

Figure 3.15: Example of the retrieval and ranking using both the interest and the time based methods.

the recommendation process, and the default value of N we have used in our approach is 3. But this value can also be defined by the developer in specific situations (see chapter 4).

The *time* based method uses the time, instead of the interest, to measure the relevance of the source code elements that are being manipulated by the developer. The interest of an element represented in the context model reflects the relevance of that element during a period of time. But, sometimes, the most relevant elements may not be those with an higher interest during that period of time, but the ones that have been accessed more recently. The time based method favors this aspect, retrieving source code elements that are related with the elements that have been manipulated more recently. The recommendations retrieved using this method include the top N elements of the context model that have been accessed more recently and all the elements that are structurally related with them. The value of N used in the time based method is the same that is used in the interest based method.

A simplified representation of the retrieval process is shown in the upper side of figure 3.15. The interest based method starts by collecting the top 3 source code elements with higher interest in the context model, shown as elements A, B and C. Then, all the elements that are structurally related with these elements are also retrieved, which are represented by elements D and E. The time based method first retrieves the 3 elements that were accessed more recently by the developer, represented as elements G, H and A, and then all the elements that are structurally related with them, represented by elements I and D. The values associated with elements A, B and C represent their interest in the structural context. The values associated with elements G, H and A represent the time elapsed since their were accessed, normalized by the time of the most recently accessed element, which is element G. The values associated with the relations represent the interest of these relations in the structural context. The number of recommendations retrieved is limited to 100, so

that the ranking of these recommendations is computed within an acceptable time frame, which was defined around $1s$.

3.4.2 Ranking

The recommendations retrieved are ranked taking into account the retrieval process and their relevance to the context model of the developer. The retrieval process is represented by an interest score (si) and a time score (st), while the relevance in relation to the context model is represented by a structural score (ss) and a lexical score (sl). The final score of a recommendation is given by a weighted sum of these scores, as shown in equation 3.25.

$$sf = (w_i \times si) + (w_t \times st) + (w_s \times ss) + (w_l \times sl) \tag{3.25}$$

$$w_i + w_t + w_s + w_l = 1 \tag{3.26}$$

The sum of the four weights is always one (see equation 3.26) and the value of each score is always normalized in the interval $[0, 1]$. Therefore, the final score of a search result is always within the interval $[0, 1]$. The structural and lexical scores are computed the same way as for the context-based search ranking, see section 3.3.2. Next, we describe in detail how the interest and time scores are computed.

Interest Score

The *interest score* represents the score of the elements that were retrieved using the interest based method. There are two types of elements retrieved, those that are in the list of the elements with higher interest in the structural context, and the ones that are structurally related with them. The elements that are retrieved in the list have a score that corresponds to their interest in the context model. The score of the elements retrieved through a structural relation with the elements in the list is computed using the interest of the relation and the interest of the element with which they are related. The interest of the element in the list is used to normalize the interest of the structural relation, so that the score of the retrieved element is proportional to the interest of the element in the list. This way, the score of the retrieved elements take into account the relevance of both the relation and the element that contributed to their retrieval. When an element has a structural relation with more than one of the elements in the list, the score is given by the average of the scores of all the relations. This way, the interest score is computed using equation 3.27, where $I(r)$ is the interest of element r in the structural context, while $I(se_i)$ and $I(sr_i)$ are the interest of the ith element and relation, respectively, that got element r retrieved.

$$si(r) = \begin{cases} I(r) & \text{if retrieved directly} \\ \frac{\sum_{i=1}^{n} I(se_i) \times I(sr_i)}{n} & \text{if retrieved indirectly} \end{cases} \tag{3.27}$$

As represented in the lower side of figure 3.15, the elements with higher interest retrieved in the top N list (A, B and C) have an interest score correspondent to their interest in the context model.

$$si(A) = 0.8$$
$$si(B) = 0.6$$
$$si(C) = 0.5$$

Then, element D was retrieved using two structural relations with elements A and C, and its score is computed as an average of the interest of these relations normalized by the interest of elements A and C.

$$si(D) = \frac{(0.8 \times 0.8) + (0.5 \times 0.6)}{2} = 0.5$$

Finally, element E was retrieved using a structural relation with element C, thus its interest score is computed by normalizing the interest of that relation with the interest of element C.

$$si(E) = \frac{0.5 \times 0.4}{1} = 0.2$$

Time Score

The *time score* represents the score of the elements that were retrieved using the time based method. The score is computed in a way that is similar to that used to compute the interest score. The main difference is that the relevance of each element is computed using the time elapsed since it was last accessed, instead of using its interest. This time span (ts) is normalized by the maximum time span (ts_{max}) among the elements in the top N list, as shown in equation 3.28.

$$T(se) = \frac{ts}{ts_{max}} \tag{3.28}$$

Similarly to the interest score, the time score is computed using equation 3.29, where $T(r)$ is the normalized time span for element r, while $T(se_i)$ is the normalized time span for the ith element and $I(sr_i)$ is the interest of the ith relation, that got element r retrieved.

$$st(r) = \begin{cases} T(r) & \text{if retrieved directly} \\ \frac{\sum_{i=1}^{n} T(se_i) \times I(sr_i)}{n} & \text{if retrieved indirectly} \end{cases} \tag{3.29}$$

As shown in figure 3.15, the time score is computed through the same rules applied to the elements retrieved using the interest based method, replacing the interest of the retrieved elements by the normalized time:

$$\begin{aligned} st(G) &= 1.0 \\ st(H) &= 0.6 \\ st(I) &= \frac{(1.0 \times 0.7) + (0.6 \times 0.6)}{2} = 0.5 \\ st(A) &= 0.3 \\ st(D) &= \frac{0.3 \times 0.8}{1} = 0.2 \end{aligned}$$

3.5 Weight Learning

As described in the previous sections, the ranking of search results and recommendations, which will be generically referred as results, is computed using a set of different components. The contribution of each component to the final ranking is defined by a set of

weights, one per each component (see sections 3.3.2 and 3.4.2). At first, these weights are equally balanced, so that each component contribute in the same proportion to the ranking of a result. We could not predict in advance which components would be more relevant in the ranking process. Furthermore, the relevance of each component could vary from developer to developer. Therefore, we have defined a learning mechanism to learn which components are more relevant for the developer, so that these components could be favored in the ranking process. This is done by learning the weights associated to each one of the components used in the ranking of the results.

This *learning* mechanism uses the results that have been selected by the developer to learn the weights that are associated to each component. This approach is based on the assumption that all the results selected by the developers can be considered useful. This way, the weights evolve based on the analysis of how each component contributed to rank the results that were useful for the developer. Because the scores of the different components are not comparable between each other, we use the ranking obtained by the result in each component to find the influence of that component in the final ranking. The final ranking of a result represents its place in the list of all results sorted by their final score, while the ranking of a component represents the place of the result when the list is sorted by the score of that component. The objective is to increase the weights of the components that contributed to promote the selected results and decrease the weights of the components that contributed to demote them.

When the developer selects a result whose ranking was computed using two or more components, the learning process is initiated. When only one component is used to rank a result, we do not have enough information to decide which components should be favored and which should not. As shown in equation 3.30, we compute the *influence* of each component (i_x) using the difference between the final ranking of the result (rf) and the individual ranking obtained for that component (rx). This way, the higher a result is ranked in a specific component, the higher is the influence of that component in the final ranking.

$$i_x = rf - rx \tag{3.30}$$

The influence obtained for each component is then normalized to the interval $[-1, 1]$, so that the components that had a higher influence get positive values, while those that had a lower influence get negative values. The *normalized influence* (ni_x) is given by equation 3.31, where i_{min} and i_{max} represent the minimum and maximum influence among all components.

$$ni_x = 2 \times \left(\frac{i_x - i_{min}}{i_{max} - i_{min}} \right) - 1 \tag{3.31}$$

Then, the positive and negative influences must be balanced, so that the weights are increased in the same proportion they are decreased, maintaining their sum as one. The *balanced influence* (bi_x), for the groups of positive (+) and negative (−) influences, is given by equations 3.32 and 3.33, where ni_t represents the sum of the influences in each group and m represents the number of components in each group.

$$bi_x^+ = \frac{ni_x^+}{ni_t^+}; \quad ni_t^+ = \sum_{k=1}^{m^+} ni_k^+ \tag{3.32}$$

$$bi_x^- = \frac{|ni_x^-|}{ni_t^-}; \quad ni_t^- = \sum_{k=1}^{m^-} ni_k^- \tag{3.33}$$

Because the learning effort needed by the system depends on how correct is the selected result, a learning coefficient is applied to the balanced influence. The value of the learning coefficient depends on the final ranking of the result that was selected by the developer. The better the ranking of the result, the lower the learning effort needed. This way, the *learning coefficient* (μ) is given by equation 3.34, where rf is the final ranking of the result and 0.01 is the maximum value for the learning coefficient. The maximum value for the learning coefficient is used to determine the global learning rate and can be adjusted to obtain a smoother or a coarser progression.

$$\mu = 0.01 \times \left(1 - \left(\frac{1}{e^{0.2 \times rf}}\right)\right) \tag{3.34}$$

Finally, the difference for each weight is obtained by applying the learning coefficient (μ) to the balanced influence (bi_x) of the corresponding component. The new weight (w'_x) is obtained by adding this difference to the previous weight, as shown in equation 3.35.

$$w'_x = w_x + (\mu \times bi_x) \tag{3.35}$$

In order to exemplify the learning mechanism described here, consider a search result selected by the developer with the following final (rf), retrieval (rr), structural (rs) and lexical (rl) rankings:

$$rf = 2; \quad rr = 6; \quad rs = 1; \quad rl = 3;$$

The influence of each component, namely retrieval (i_r), structural (i_s) and lexical (i_l), would be computed using equation 3.30, as follows:

$$\begin{aligned} i_r &= rf - rr = 2 - 6 = -4 \\ i_s &= rf - rs = 2 - 1 = 1 \\ i_l &= rf - rl = 2 - 3 = -1 \end{aligned}$$

The influence of each component would then be normalized to the interval $[-1, 1]$, so that the components that contributed to promote the result get values closer to 1, while those that contributed to demote it get values closer to -1. The normalized influence is computed using equation 3.31, as follows:

$$\begin{aligned} ni_r &= 2 \times \left(\frac{(-4) - (-4)}{1 - (-4)}\right) - 1 = -1 \\ ni_s &= 2 \times \left(\frac{(1) - (-4)}{1 - (-4)}\right) - 1 = 1 \\ ni_l &= 2 \times \left(\frac{(-1) - (-4)}{1 - (-4)}\right) - 1 = 0.2 \end{aligned}$$

The normalized influences are then balanced, so that the positive and negative influences become symmetric and their sum is zero. Following our example, the retrieval influence belongs to the group of negative influences, while the structural and lexical influences belong to the positive ones. The total influence for each group is computed as a sum of their influences:

$$\begin{aligned} ni_t^- &= -1 \\ ni_t^+ &= 1 + 0.2 = 1.2 \end{aligned}$$

The influences are balanced using equations 3.32 and 3.33, as follows:

$$
\begin{aligned}
bi_r^- &= \frac{|-1|}{-1} = -1 \\
bi_s^+ &= \frac{1}{1.2} \approx 0.8 \\
bi_l^+ &= \frac{0.2}{1.2} \approx 0.2
\end{aligned}
$$

The learning coefficient is computed using the final ranking of the result (rf), which is 2 for our example. Because the final ranking is low, thus the result was well ranked, the learning coefficient will be less significant. Using equation 3.34, the learning coefficient is computed as follows:

$$
\mu = 0.01 \times \left(1 - \left(\frac{1}{e^{0.2\times 2}}\right)\right) \approx 0.003
$$

Finally, the new weights for the retrieval (w'_r), structural (w'_s) and lexical (w'_l) components would be computed using equation 3.35, by summing the balanced influence of each component to its previous weight, as follows:

$$
\begin{aligned}
w'_r &= 0.33 + (0.003 \times -1) = 0.3270 \\
w'_s &= 0.33 + (0.003 \times 0.8) = 0.3324 \\
w'_l &= 0.33 + (0.003 \times 0.2) = 0.3306
\end{aligned}
$$

As demonstrated by the previous example, the result that was selected by the developer was much better ranked with the structural component ($rs = 1$), and even with the lexical component ($rl = 3$), then with the retrieval component ($rr = 6$). Therefore, the weights of the structural and lexical components were slightly increased, while the weight of the retrieval component was decreased. The difference in the weights was minimal because the result was already well ranked ($rf = 2$), the difference would be higher if the result was poorly ranked.

3.6 Summary

We started this chapter by describing a broader perspective of the work environment of a software developer, where the different dimensions that comprise her/his work are presented in a layered model. This model includes a *personal layer*, a *project layer*, an *organization layer* and a *community layer*. The contextual information of the developer, at each one of these layers, can be used to improve the retrieval of information that is relevant for her/his work. The research work described in this thesis is focused on the personal layer of this model, more specifically, we have focused on how the contextual information of the developer could be used to improve the search and recommendation of source code in the IDE. Then we have presented the various mechanisms that comprise the basis of our approach, including the knowledge base, the context model, the context-based search and recommendation processes, and the learning of the weights used in these approaches.

The *knowledge base* represents the source code structure that is stored in the workspace of the developer. The source code structure is formalized using a structural ontology, representing the source code elements and their structural relations, and a lexical ontology,

representing the terms used to reference the source code elements and how these terms are associated. The knowledge base is build through various steps, including the parsing of the source code, the disambiguation of source code references, the extraction of terms from the source code, and the population the two ontologies with the information extracted in the previous steps. Finally, the source code elements represented in the knowledge base are also indexed for later retrieval.

The contextual information of the developer is modeled using a *context model*, that is built from the interactions of the developer with the source code elements in the workspace. As in the knowledge base, this context model combines a structural and a lexical dimensions, which represent the source code elements, their structural relations and terms that are more relevant for the developer in a specific moment in time. A context transition detection mechanism allows the context model to automatically adapt to the changes in the focus of attention of the developer.

The context model defined is used to improve the ranking of source code elements retrieved using a *context-based search* process. The retrieval is performed using an IR model, by matching the terms in the identifiers of the source code elements with the terms contained in the search query. The retrieved elements are then ranked according to a retrieval, a structural and a lexical components. The retrieval component represents the ranking provided by the IR model, while the structural and lexical components represent the proximity of the search result to the context model of the developer.

The same context model was used to support the *context-based recommendation* of relevant source code elements to the developer. The recommendations are retrieved using the source code elements with higher interest and accessed more recently, that are represented in the context model. The ranking of recommendations takes into account an interest and a time components, representing the ranking obtained through the retrieval process, as well as a structural and lexical components, which represent the proximity of the recommendation in relation to the context model.

The contribution of the different components to the ranking of search results and recommendations is defined by a set of weights that are learned over time. The best combination of weights is inferred using a *learning* mechanism based on the search results and recommendations selected by the developer. The objective of this mechanism is to increase the weights of the components that contributed to promote the selected results and decrease the weights of the components that contributed to demote them.

In the following chapter, we describe the prototype implemented, which allowed us to integrate our approach into the Eclipse IDE. We start by describing the various elements that comprise the system architecture. Then, we focus on the features that are provided to a developer using the prototype.

Chapter 4

Implementation

"Genius is one percent inspiration, ninety-nine percent perspiration."

Thomas Edison

The context-based search and recommendation approaches described in the previous chapter were integrated in the Eclipse[1] IDE, using a prototype plugin named Software Development in Context (SDiC). The plugin automatically builds and maintains the knowledge base, updating the structural and lexical ontologies whenever source code base changes. The context model is automatically captured from the interactions of the developer in the IDE. These operations are run in background, being completely hidden from the developer and not requiring any kind of human intervention. The context-based search and recommendation of source code elements are accessible through specific interfaces that were added to the Eclipse IDE. The plugin was implemented using the Java[2] programming language, as required by the Eclipse platform. Next, we describe the different components that comprise the implementation architecture of the prototype and explain in more detail the features that are provided to the developer.

4.1 Architecture

The architecture of the prototype implemented comprises a *data layer*, a *business layer* and a *presentation layer*, as illustrated in figure 4.1. The different components that comprise each one of these layers are described in the following sections.

4.1.1 Data Layer

The *data layer* comprises the storage mechanisms necessary for the system operation, including the Knowledge Base and the Database. The first is used to store the knowledge needed for the system to operate, while the later is used to store the system usage data.

Knowledge Base The Knowledge Base stores the ontologies used in the system and indexes the source code elements. The structural and lexical ontologies were stored and manipulated as a graph using Neo4J[3], a high-performance graph database that allowed

[1] `http://eclipse.org` (August 2012)
[2] `http://www.oracle.com/us/technologies/java/` (August 2012)
[3] `http://neo4j.org/` (August 2012)

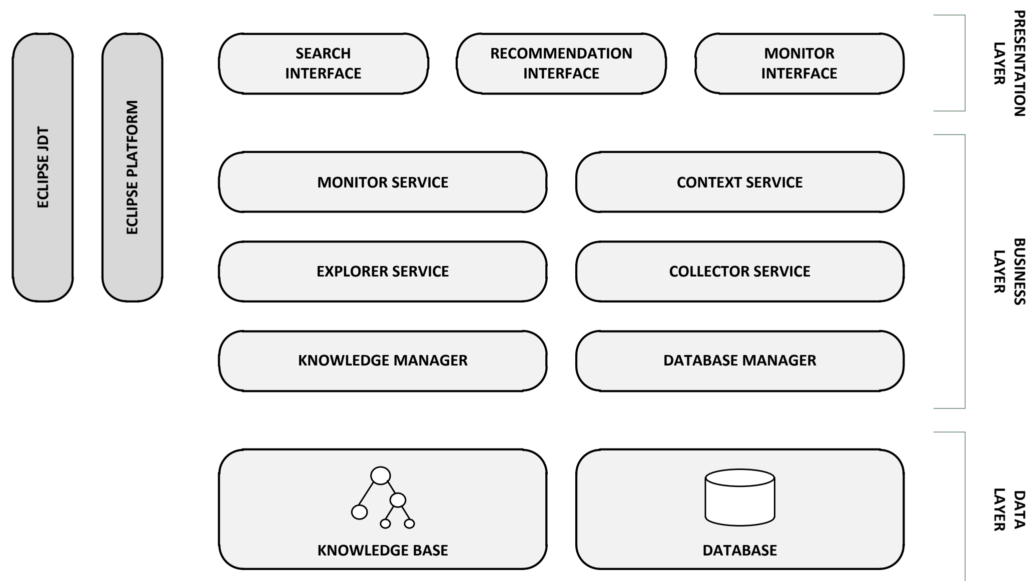

Figure 4.1: The architecture of the prototype implemented.

us to efficiently store and manipulate the large number of elements and relations represented in these ontologies. The indexing and retrieval mechanisms are provided by Apache Lucene[4], a high-performance, full-featured text search engine. This engine uses a Vector Space Model (VSM) approach based on Term Frequency/Inverse Document Frequency (TF-IDF). The term weighting and document ranking process used in Apache Lucene has some refinements[5], which are meant to improve search quality and usability and that are not relevant in the context of our approach.

Database The Database stores system usage data that is collected for analysis, including statistical information about the knowledge base, context model, search process and recommendation process. The validation results were extracted from the data stored in this database. This collected data is stored in a H2[6] database engine, embedded within the system.

4.1.2 Business Layer

The business layer comprises the main modules of the system, which provide the mechanisms to interact with the data structures of the data layer and the algorithms necessary to deliver the features offered through the presentation layer.

Knowledge Manager The Knowledge Manager module provides the operations necessary to manage the knowledge stored in the Knowledge Base. These operations include adding, updating and deleting information, as well as indexing and retrieving the source code elements represented in the knowledge base. The search and recommendation mechanisms are also implemented in this module, including the retrieval and ranking processes.

[4] `http://lucene.apache.org/` (August 2012)

[5] `http://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/api/core/org/apache/lucene/search/Similarity.html` (August 2012)

[6] `http://www.h2database.com/` (August 2012)

Database Manager The Database Manager module encapsulates the Database, providing the operations needed to manage the data collected about the system usage.

Collector Service The Collector Service module deals with the process of collecting the system usage data and uploading this data to a server. The process can be performed manually by the developer, or automatically by the system, on a daily basis.

Explorer Service The Explorer Service module provides an interface for accessing the Java source code files in the Eclipse workspace and to parse the Java source code. These operations are supported by the Eclipse Java Development Tools (JDT)[7], which provides the necessary plug-ins to create a Java IDE out of the Eclipse platform. The JDT tools allow access to a model representing the entire Java source code structure, from top level projects to individual source code elements. It also provides the necessary mechanisms to parse a Java source code file into an Abstract Syntax Tree (AST)[8].

Monitor Service The Monitor Service module is responsible for monitoring the changes applied to the source code files in the workspace and the interactions of the developer with such files. These changes and interactions are captured through the Eclipse Platform[9], which provides the necessary mechanisms to detect these activities. The information about the activity of the developer is made available to other modules through an event-based mechanism. When a change in the workspace or an interaction of the developer is registered, a corresponding event is fired to all the modules that are registered as consumers of that event.

Context Service The Context Service module maintains a representation of the contextual information of the developer. This module consumes the interaction events produced by the Monitor Service module, from which a context model is built. Each interaction is processed in order to update the current context model, or to make a context transition, when that is the case. The various context models that may exist in a specific moment are stored in a context model pool. Although only one of them is active at each moment, the system may make a transition to an existing context model in certain conditions. The changes applied to a context model, or to the context model pool, produce events that can be captured by other modules that may need to be updated as a result of these changes.

4.1.3 Presentation Layer

The presentation layer comprises the user interfaces provided by the system, including the Search Interface, the Recommendation Interface and the Monitor Interface. These interfaces are fully integrated in the Eclipse environment and are the contact point between the developer and the features provided by the system.

Search Interface The Search Interface provides an interface for the context-based search process. This interface allows the developer to submit a query and explore the search results. The search interface is described in more detail in section 4.2.1.

Recommendation Interface The Recommendation Interface provides an interface for the context-based recommendation process. This interface allows the developer to explore the recommendations provided by the system. The recommendation interface is described in more detail in section 4.2.2.

[7] `http://www.eclipse.org/jdt/` (August 2012)
[8] `http://en.wikipedia.org/wiki/Abstract_syntax_tree` (August 2012)
[9] `http://www.eclipse.org/platform/` (August 2012)

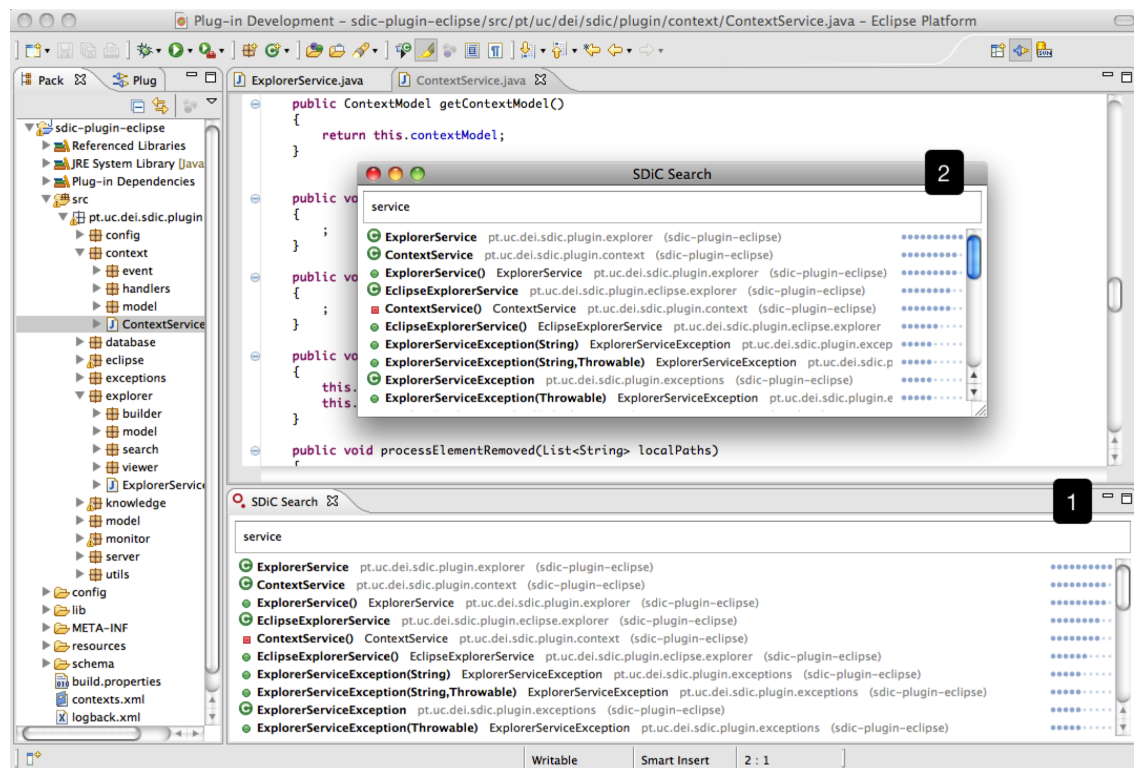

Figure 4.2: A screenshot of the prototype showing the search view (1) and the search window (2).

Monitor Interface The Monitor Interface interface is used to observe some information about different aspects of the system, including the knowledge base, the context model and the retrieval processes. The monitor interface is described in more detail in section 4.2.3.

4.2 Features

The main features provided to the developer by our prototype include interfaces for *search* and *recommendation* of source code elements, which are based on the context-based search and recommendation methodologies developed. Additionally, the prototype provides an interface to *monitor* a set of information related to the system operation. These features are described in more detail in the following sections.

4.2.1 Search

The *search* feature is available through a specific interface, that can be used as an Eclipse View or an independent window. The search view (see 1 in figure 4.2) can be easily integrated in the main window of the IDE, while the search window (see 2 in figure 4.2) can be triggered at any moment, using a combination of keys (Ctrl+Alt+S), allowing the developer to perform the whole search process using only the keyboard. The search interface uses an *incremental search*[10] mechanism, providing search results promptly. As the developer writes the search query, the search interface is continually updated with the to 30 search results for the current query. We believe that the combination of an

[10] http://en.wikipedia.org/wiki/Incremental_search (August 2012)

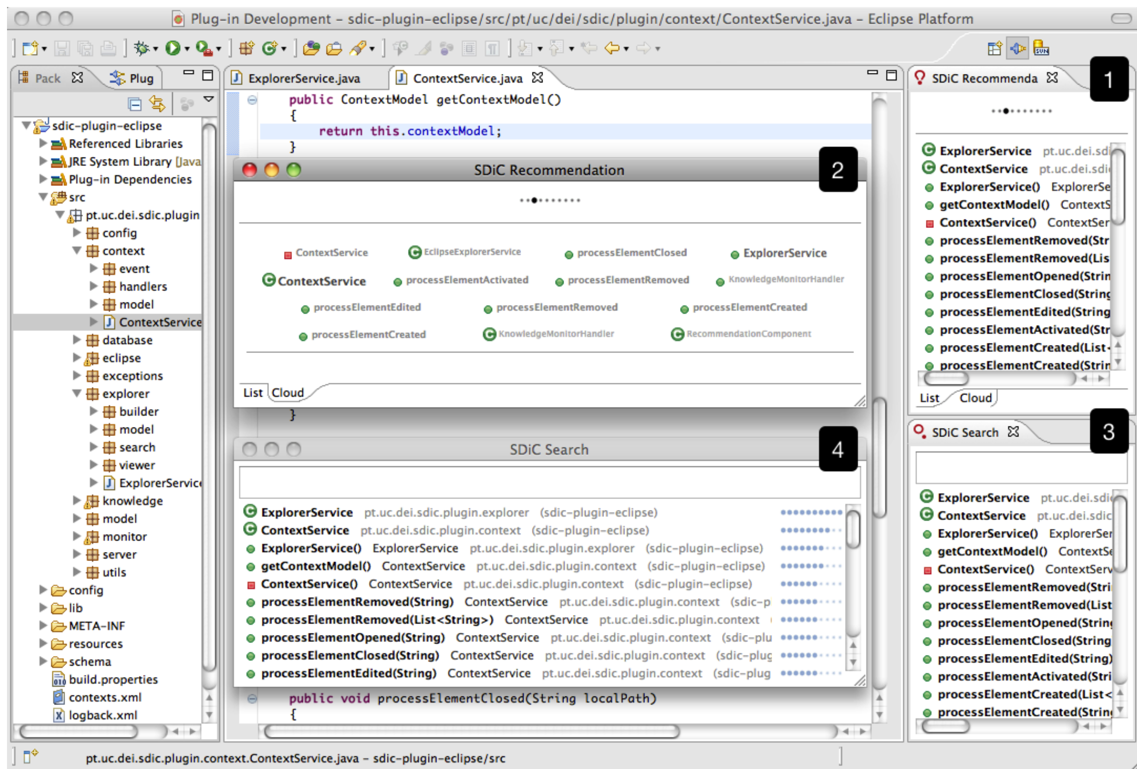

Figure 4.3: A screenshot of the prototype showing the recommendation view (1), the recommendation window (2), and the integration of recommendations in the search interfaces (3 and 4).

easy to use interface with a more precise search, makes our prototype a great tool to help developers reach the desired source code elements quickly and efficiently.

Based on our experience, search is something that developers use when they do not exactly know what they are looking for, or when what they want is somehow hidden in the complexity of the source code structure. But, our prototype can also be used when the developers know exactly what they are looking for, avoiding the need to spend time browsing for a source code element in the source code structure or in the source code files. Because the search results are ranked according to the context of the developer, most of the times a query of a few characters is enough for the desired source code element to appear in the first search results. The use of the context of the developer allows us to identify what is more relevant for the developer among all the possibilities, and that is what distinguish our approach from the traditional search tools provided by the Eclipse IDE.

4.2.2 Recommendation

The *recommendation* feature provides recommendations of relevant source code elements through a specific interface, that can be used as an Eclipse View or an independent window. The recommendation view (see 1 in figure 4.3) can be integrated in the main window of the IDE and stay visible all the time to keep the recommendations easily accessible. The recommendation window (see 2 in figure 4.3) is an alternative that spares space in the IDE, it can be triggered using a combination of keys (Ctrl+Alt+R) and is fully functional using only the keyboard. This interface provides recommendations in the form of a list and in what we call a *code cloud* (see 2 in figure 4.3). The recommendations

are automatically updated as the developer interacts with the IDE, reflecting the changes that occur in the context model. The top 30 recommendations with higher score are presented to the developer each time the the context model changes. The number of elements of the structural context that are used for the recomendation process, which we call of N (see section 3.4.1), is 3 by default, but can be set by the developer between 1 and 10, and is represented by the dotted scale in the top of the interface. By providing an ordered list of source code elements that may be of interest to the developer in a specific moment, we aim to decrease the effort needed to access such elements. This list provides a shortcut to jump to the desired source code elements and helps the developers maintaining awareness of what is more important in their current context. The recommendations are also integrated with the search interface and are automatically shown to the developer before performing the search (see 3 and 4 in figure 4.3). This integration is based on the assumption that if the recommendations include the source code element desired by the developer, this will avoid the need to perform the search. The value of N used for the recommendations shown in the search interface is always 3, for reasons of simplicity of the search interface.

4.2.3 Monitor

The *monitor* feature comprises an interface where a set of information related to the system operation is provided (see figures 4.4 to 4.9). This interface was especially implemented so that we could analyze important information about the system behaviour in different situations. Eventually, it was also made available to the users, and turned out to be an interesting way of explaining the concepts behind the system and motivating its use. The interface is divided in four sections, accessible through different tabs, as shown in the bottom of figure 4.4, which provide information related to the context model, the knowledge base, the search feature and the recommendation feature. The charts shown in these interfaces were created using JFreeChart[11], a chart library for Java.

The information related with the context modeling and context transition processes can be viewed in tab Context Model, as shown in figures 4.4 and 4.5. The interface shows a list of all the context models that have been created and which one of them is currently active (see 1 in figure 4.4). When one of these context models is selected, all the information related with the selected context model is presented, including information about the structural and lexical contexts. Also, a list of events (Added, Removed, Activated and Deactivated) associated with the context model is presented (see 2 in figure 4.4). Concerning the structural context, we can see the structural elements (see 3 in figure 4.4) and structural relations (see 4 in figure 4.4) that are currently included in the selected context model, as well as a visual representation of their current interest value. The hard and soft transitive elements are shown in gray, with the soft transitive elements having a lighter grey than the others. When one of these elements, or relations, is selected, a list of the events (Added, Removed, Interest Incremented and Interest Decremented) that affected that element are presented (see 5 in figure 4.4) and a chart representing the evolution of its interest value over time is shown (see 6 in figure 4.4). Concerning the lexical context model, we can see the terms that exist in the current context model (see 1 in figure 4.5). Similarly to the structural context interface, when a term is selected, the list of events that affected the term is presented (see 2 in figure 4.5) and a chart representing the evolution of its interest value over time is shown (see 3 in figure 4.5).

The information about the knowledge base is available through the tab Knowledge Base, as shown in figures 4.6 and 4.7. With respect to the structural ontology, the number of instances per each sub-class of structural element (Class, Interface and Method) is presented

[11] `http://www.jfree.org/jfreechart/` (August 2012)

using a pie chart (see 1 in figure 4.6). The number of structural relations (extensionOf, implementationOf, attributeOf, methodOf, parameterOf, returnOf, calledBy and usedBy) that were created between these instances is presented using a similar chart (see 2 in figure 4.6). The evolution in the number of elements represented in the structural ontology over time is described using a historical chart (see 3 in figure 4.6), and a similar chart is used to present the number of relations (see 4 in figure 4.6). Concerning the lexical ontology, the current number of terms represented in the knowledge base and its evolution over time are presented using a historical chart (see 1 in figure 4.7). The number of lexical relations that were created between terms, and between terms and structural elements, are presented using a pie chart (see 2 in figure 4.7), and its evolution is also presented using a historical chart (see 3 in figure 4.7).

The information about the search results selected by the developer is accessible through the tab Search, as shown in figure 4.8. The distribution of the selected search results per each type of interface (Search View and Search Window) is presented using a pie chart (see 1 in figure 4.8), while the evolution of the average rankings of these results, per ranking component (Retrieval, Structural and Lexical), is presented using a historical chart (see 2 in figure 4.8). The current weight for each one of the ranking components is also presented (see 3 in figure 4.8), along with a historical chart showing their evolution over time (see 4 in figure 4.8).

The information about the recommendations that were selected by the developer are provided through tab Recommendation, as shown in figure 4.9. This interface is very similar to that used to show information related to the selected search results. The number of recommendations selected per interface (Recommendation View and Recommendation Window), including the search interfaces (Search View and Search Window), are presented using a pie chart (see 1 in figure 4.9). The evolution observed in the average rankings of the selected recommendations, per ranking component (Interest, Time, Structural and Lexical), is shown using a historical chart (see 2 in figure 4.9). The current weights of these ranking components and their evolution are also presented (see 3 and 4 in figure 4.9).

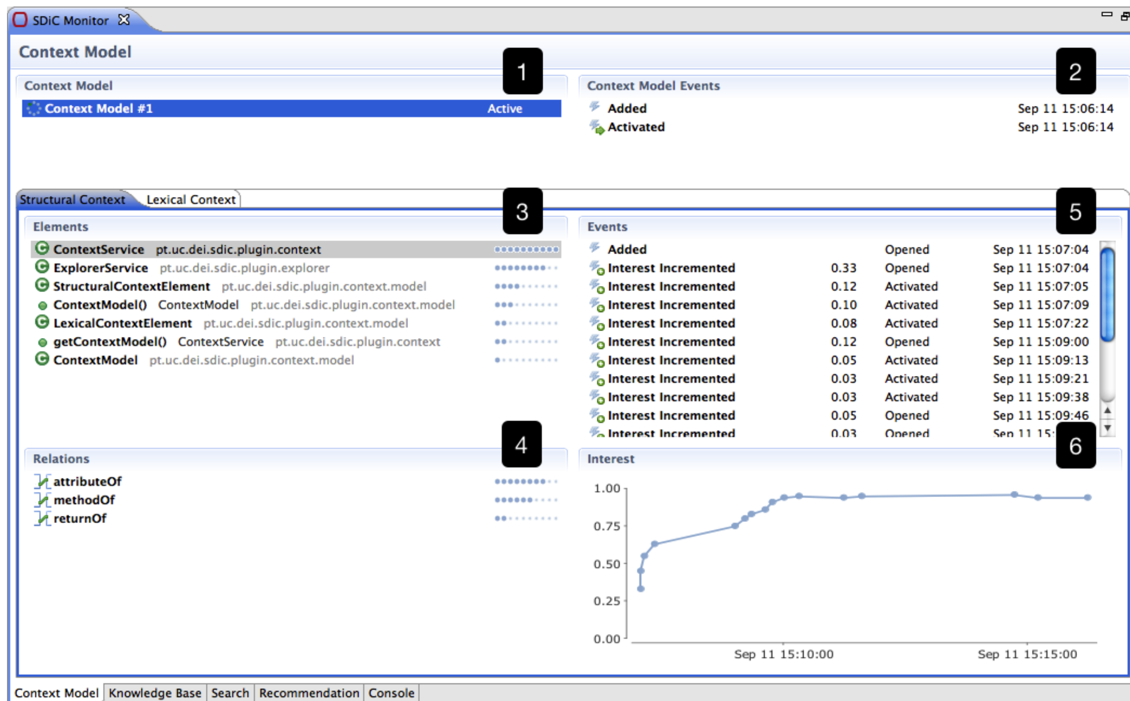

Figure 4.4: A screenshot of the monitor interface showing information about the structural context, including the list of existing context models (1), the list of events associated to the current context model (2), the list of structural elements (3), the list of structural relations (4), the list of events associated to the current structural element (5), and the evolution of the interest of the current structural element (6).

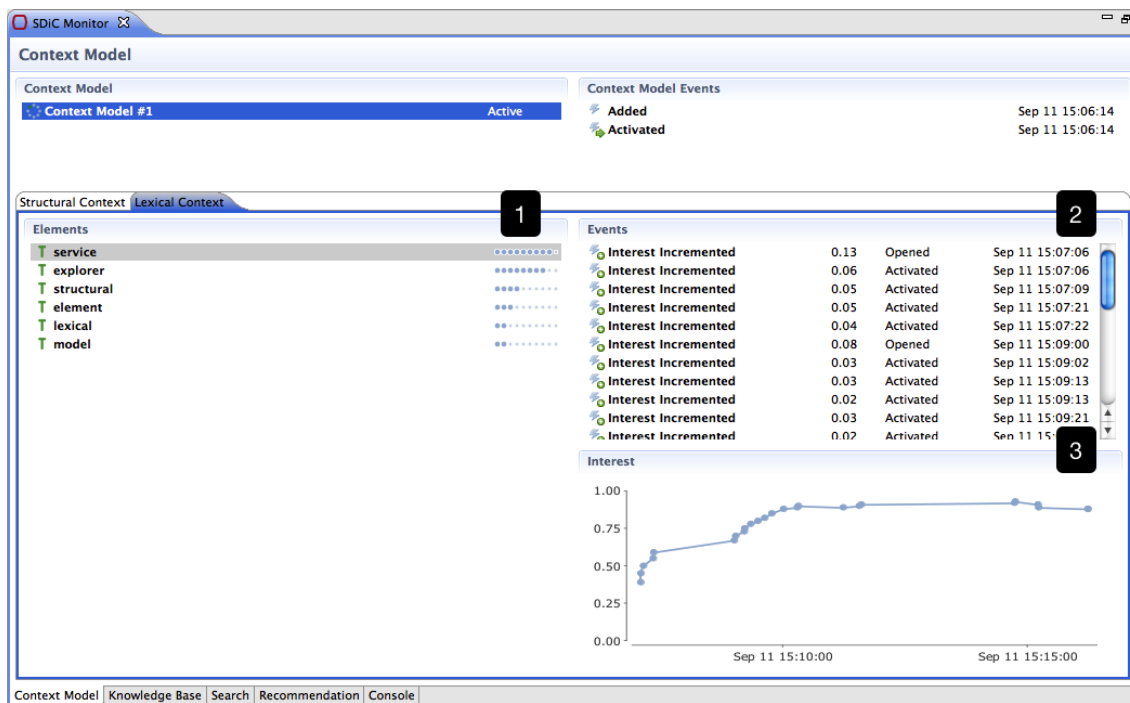

Figure 4.5: A screenshot of the monitor interface showing information about the lexical context, including the list of terms (1), the list of events associated to the current term (2), and the evolution of the interest of the current term (3).

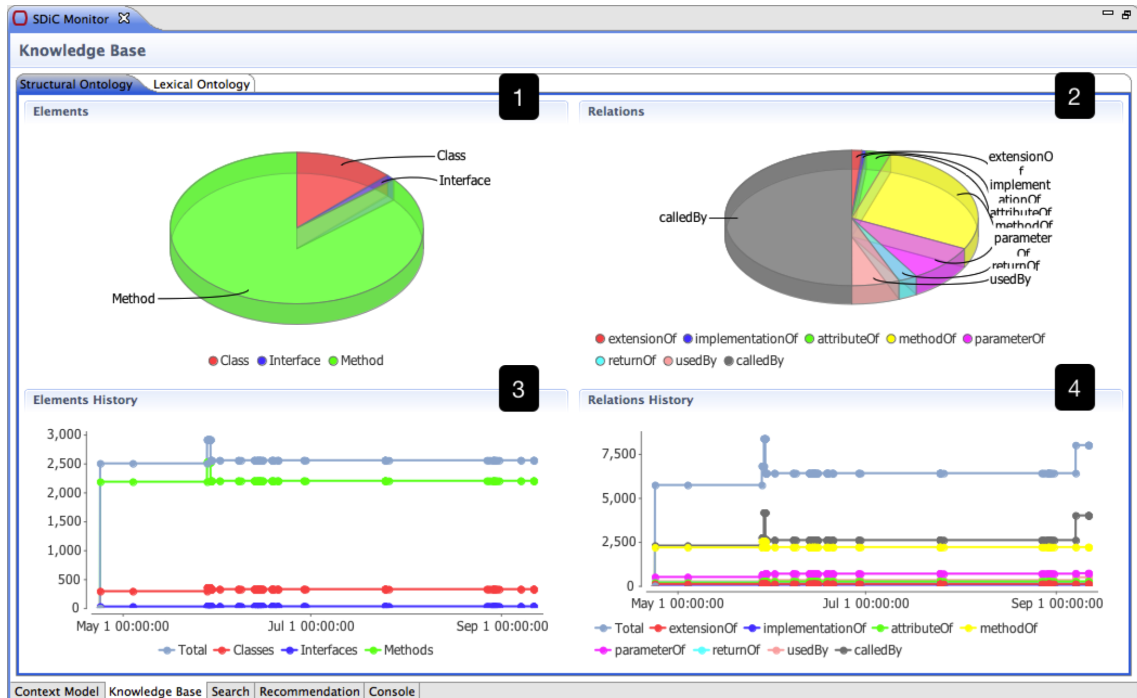

Figure 4.6: A screenshot of the monitor interface showing information about the structural ontology, including the number of structural elements (1), the number of structural relations (2), the evolution in the number of structural elements (3), and the evolution in the number of structural relations (4).

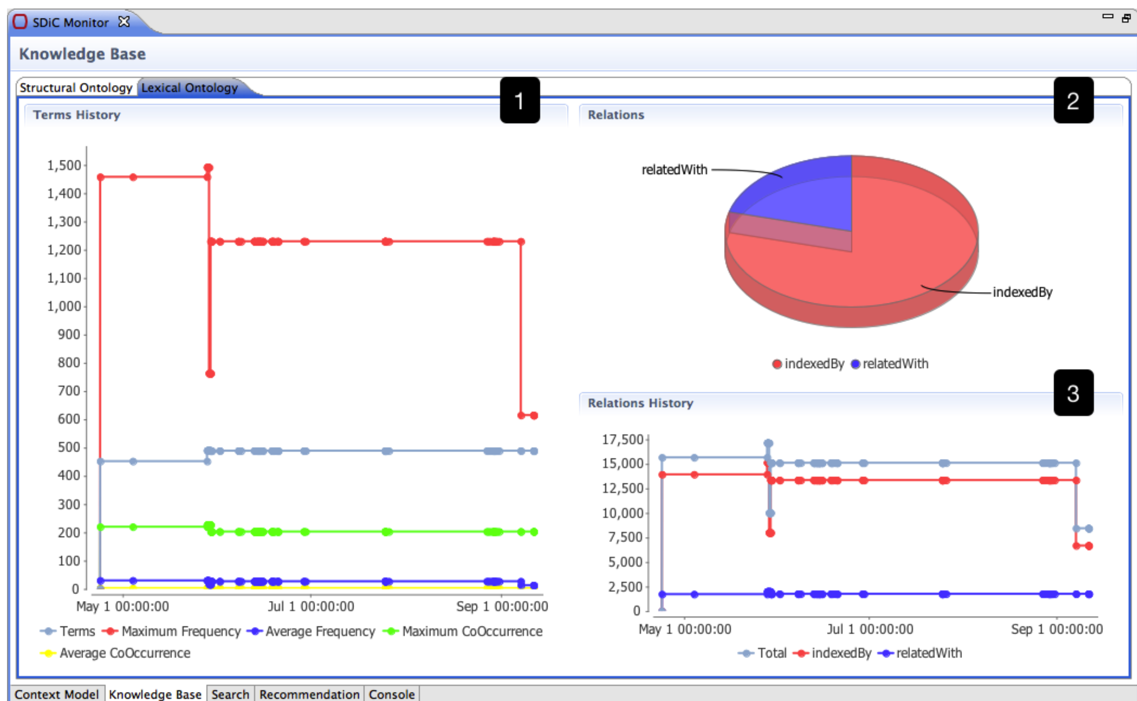

Figure 4.7: A screenshot of the monitor interface showing information about the lexical ontology, including the evolution in the number of terms (1), the number of lexical relations (2), and the evolution in the number of lexical relations (3).

Figure 4.8: A screenshot of the monitor interface showing information about the context-based search, including the number of selected search results (1), the distribution of the search weights (2), the evolution of the average rankings of selected search results (3), and the evolution of the search weights (4).

Figure 4.9: A screenshot of the monitor interface showing information about the context-based recommendation, including the number of selected recommendations (1), the distribution of the recommendation weights (2), the evolution of the average rankings of selected recommendations (3), and the evolution of the recommendation weights (4).

Chapter 5

Validation

"The pure and simple truth is rarely pure and never simple."

Oscar Wild

This chapter presents the process that we have followed to validate our approach and discusses the results we have collected during that process. When evaluating an approach based on something so complex and subjective as context, it is difficult to achieve results using a laboratory controlled experiment. The context of a developer depends on an unpredictable number of factors, and it would be almost impossible to simulate the different conditions that influence the context of different developers in a laboratory environment. Therefore, we believe that our approach had to be validated with developers working on a real world scenario. These real world experiments pose several challenges and are difficult to implement at a large scale, because it is difficult to gather a significant number of developers that are available to install and evaluate an experimental prototype during their work. Nevertheless, we managed to run our experiments with a reasonable number of developers. We started by conducting a preliminary study with a smaller number of developers. This study was essential to mature our approach, by identifying a set of issues that could be improved. The lessons learned during the preliminary study allowed us to improve our approach and conduct a new study, this time with an higher number of developers. The two studies and their results are described in the following sections. The chapter concludes with a discussion of the results obtained during the validation process.

5.1 Preliminary Study

A preliminary study was created to verify if our approach could be used in a real scenario and to gain insights about what we could be doing wrong and could be improved. This preliminary study was conducted with an initial version of the prototype that had some differences in relation to our final version, which was implemented based on the approach described in chapter 3. Before discussing the results obtained for the context-based search and recommendation in these experiments, we describe the reasons for the differences between the initial and final versions of the prototype, as follows.

User Interface. The first user interface implemented was simpler and did not include much of the useful features that can be found in the final version. Most of the improvements were added later, based on the feedback we have collected from the developers in regular and informal talks maintained during this preliminary study. Although our focus was not in the user interface, the developers were using the prototype during their work

and they were quite sensitive to usability issues. These improvements were essential for the acceptance of the prototype by the developers, otherwise it would have been neglected and we would not be able to validate our approach. We tried to solve the problems as soon as they were detected, and most of them were solved even before the end of the preliminary study. Here we will describe only the most important improvements that led to the final user interface. There were other suggestions from the developers that we have not implemented, either because they were outside the scope of this work, or because they required an implementation effort for which we had no capacity.

- The initial version of the prototype included only the search interface in the form of an Eclipse View, which was integrated in the Eclipse main window. Some developers said this view was occupying too much space in their workspace, which was already populated with other views. Furthermore, when they had the source code editor in full screen, the search view was hidden and it was difficult to access it. These concerns were addressed with the implementation of the search interface in the form of an independent window. This window could be easily accessed using a combination of keys and allowed the developers to use the search feature independently of the configuration used in their workspace.
- Because developers spend most of their time using the keyboard, we realized that the search process should not require them to use the mouse. They should be able to perform a search, jump to the desired result and continue their work, without taking their hands off the keyboard.
- The initial interface also allowed the developers to view the search results categorized by type, list of packages and hierarchy of packages. We realized that these features were almost ignored by the developers, because they wanted the process to be simple and fast. Following this idea, we removed the categorization options and reduced the interface to a list of search results. The search interface was transformed into an incremental search, so that the list of search results could be automatically gathered while the developer was typing the query. This new interface was cleaner and faster to use, but then the developers had some difficulties in distinguishing the search results, because there was results with very similar, or even equal, names. This problem was easily solved by including additional information associated to each search result, namely the type, package and project to which the search result belonged.
- When we introduced the recommendation features, in the second experiment of the preliminary study, it was accessible through a separate window. The developers soon started complaining that it was almost impossible to use, because they would not bother opening a window just to look for something that might not even be there. They suggested that it would make more sense to integrate the recommendations in the search interface, so that they would have to deal with only one window. Also, if the desired source code element was recommended, they would not have to perform a search.

Knowledge Base. The initial knowledge base needed to be complemented with some additional information, both in the structural ontology and in the lexical ontology. The structural ontology was similar to the final version, but was lacking the behaviour relations, namely calledBy and usedBy. They were not included in our first implementation because they would require an extra processing time during the parsing of the source code elements. But we realized that these relations were essential to detect the structural proximity between the source code elements. Actually, after being extracted, these relations represent the great majority of the structural relations in the knowledge base. The lexical

ontology structure was the same as in the final version, but the statistical information about term and co-occurrence frequencies was not being stored. The term frequencies allowed us to identify the very frequent terms that were distorting the metric used to find the proximity between terms. The co-occurrence frequency was used to assign a weight to the co-occurrence relations, which were later applied to compute the cost of the paths between terms.

Context Model. In its initial version, the context model did not include the structural relations, which were latter added to the structural context and associated to an interest value that is used to compute the cost of the paths between source code elements. Also, in this initial version we did not have a mechanism to detect context transitions. That mechanism was introduced to address one problem that was reported by some developers when they started using the recommendation feature. They observed that when they changed their focus of attention to a different part of the source code, the recommendation feature would continue recommending source code elements related to their previous focus of attention and would took some time to adapt to the new situation. They have realized this because the recommendation feature is more dependent on the context model than the search feature. It relies entirely on the context model to identify the source code elements that may be relevant for the developer, whereas the search feature uses a query provided by the developer. We noticed that our context model was not able to adapt to this sudden changes in the focus of attention of the developer. Thus, we have created the context transition mechanism to detect these changes and adapt to them.

Context-Based Search/Recommendation. Because of the lack of information in the initial versions of the knowledge base and the context model, the ranking of the results was slightly different. The structural score was computed the same way as in the final version, but the cost of the structural relations was fixed, instead of being computed using the interest associated to these relations in the structural context. The fixed cost assigned to these relations were solely based on our interpretation and experience, thus it would always be subjective. This way, we decided to associate the cost of such relations to the interest they could have in the context of the developer. The lexical score was also computed the same way, but by that time the cost of the co-occurrence relations was fixed and the most frequent terms were not being ignored in the paths between terms. The statistical information added to the knowledge base allowed us to associate the cost of a co-occurrence relation to the weight of that relation. Also, we were able to ignore the very frequent terms that were creating too much paths between the terms and distorting our metric.

Learning. The learning of the weights associated to the different components were similar to the final version, but the coefficient value was fixed and did not depend on the ranking of the final result. Because of this, a result that was ranked in first place would lead to a change in the weights on the same scale as another result ranked, for instance, in tenth place. This was solved by using a learning coefficient that is higher for poorly ranked results and lower for well ranked ones.

The preliminary study had two phases. A first experiment, called of experiment A, was intended to test only the context-based search and was conducted with a team of 5 developers from a software house. The second experiment, called of experiment B, included the context-based search and recommendation and was conducted with 10 developers, 4 from a software house and 6 from a computer science graduation. The 4 developers from a software house in experiment B were also in experiment A.

The developers were using the Eclipse IDE to develop source code in the Java programming language. The experience of the developers with Java was diverse, ranging from

3 years up to more than 10, with an average around 6 years, and they have been using Eclipse for different periods of time, ranging from 3 years up to more than 10, with an average around 5 years. In average, for the two experiments, the knowledge base of each developer contained 3,094 structural elements, 3,753 structural relations, 543 lexical elements and 5,136 lexical relations. The application was installed in the work environment of the developers and presented as an innovative approach to retrieve source code artifacts in the IDE. They were asked to use the application for about three weeks in each phase. We have collected quantitative and qualitative results in the two experiments that are described in the following sections. These will be analyzed according to the search and recommendation features.

5.1.1 Context-Based Search

Concerning the context-based search, we wanted to find evidence that the use of the context model would improve the ranking of relevant search results, reducing the effort of the developers on finding the desired source code artifacts. This way, we had to verify if the use of the context model was having a positive impact on the search process. We also wanted to collect the opinions of the developers in relation to the prototype developed and identify anything we could be doing wrong. The quantitative and qualitative results collected are discussed in the following sections.

Quantitative Results

The impact of the context in the search process could be evaluated in two ways, one by analyzing the ranking of the search results that were selected by the developers, other by analyzing the evolution of the search weights as they adapt to the behavior of the developers. With regard to the ranking of the search results, the final ranking depends on the combination of three components: retrieval, structural and lexical. The retrieval component represents a typical keyword-based search process, that is not influenced by the context model and can be used as a reference ranking. The other two components represent the influence of the context model in the final ranking of a search result. During the experiment, we stored information about the search results that were selected by the developer, namely the ranking obtained by the search result in each one of the three components that contribute to the final ranking. The best ranking would be 1 and the worst ranking would be 30, since only the top 30 search results are presented to the developer. The data collected is summarized in table 5.1. The total number of search results selected by developers was 720, from which 335 were selected during experiment A and 385 during experiment B. In average, the search results selected were ranked in 2.80 place, with a retrieval ranking of 9.79, a structural ranking of 2.70 and a lexical ranking of 5.78. These values clearly indicate that the search results that were relevant for the developer were frequently better ranked through the context components than through the keyword-based process. This behaviour shows evidence that the context components had a positive influence in the final ranking of these search results, which would be ranked in much lower positions if a simple keyword-based retrieval process was used.

The contribution of the retrieval, structural and lexical components for the final ranking is defined by a set of weights that are learned from the behaviour of the developer. The evolution of these weights reflect the importance of each component for the developer, because each weight is increased or decreased according to the influence of the respective component in the ranking of search results that were relevant for the developer. The final weights for each component are here compared using a weighted mean, instead of a simple mean, because the weights change according to the search results selected by

Table 5.1: The mean and confidence interval for the rankings of the selected search results, per component and experiment.

	A		B		A+B	
	$\overline{x}$	**CI (95%)**	$\overline{x}$	**CI (95%)**	$\overline{x}$	**CI (95%)**
Retrieval Rank (rr)	10.69	± 1.01	9.00	± 0.94	**9.79**	± **0.69**
Structural Rank (rs)	2.88	± 0.46	2.55	± 0.36	**2.70**	± **0.29**
Lexical Rank (rl)	5.98	± 0.75	5.58	± 0.66	**5.78**	± **0.50**
Final Rank (rf)	2.57	± 0.30	3.01	± 0.38	**2.80**	± **0.25**

Table 5.2: The weighted mean for the final weights of the context-based search, per component and experiment.

	A		B		A+B	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
Retrieval Weight (w_r)	0.096	± 0.069	0.121	± 0.109	**0.109**	± **0.093**
Structural Weight (w_s)	0.552	± 0.092	0.565	± 0.156	**0.559**	± **0.130**
Lexical Weight (w_l)	0.318	± 0.061	0.337	± 0.052	**0.328**	± **0.057**

the developer. Thus, the weights of the developers that selected more search results have more relevance for the mean, than those of the developers that selected less results. This way, the weighted average we use takes into account the number of selected results by the developer to compute the average final weights among all the developers. The weighted mean of the final weights, per experiment, are presented in table 5.2, resulting in an average retrieval weight of 0.109, an average structural weight of 0.559 and an average lexical weight of 0.328. As expected, the final weights confirm the tendency for a growth in the contribution of the context components over the retrieval component. This growth was notable and consistent in the two experiments, with a predominance of the structural component over the lexical one. Once more, this evolution reflect the importance of the context components over the retrieval component.

Qualitative Results

By the end of the experiment, developers were asked to fill a questionnaire. The objective of the questionnaire was to perceive the opinion of the developers on the utility and quality of the application. We also wanted to know what they liked the most, and the least, and what suggestions they would give to improve the application. The questions and results of the questionnaire are shown in table 5.3. In general, the answers were very positive. Concerning the utility and usability of the application, the results show that it was considered useful and with good usability. The impact of the application on the productivity of the developers was also rated positively. The search results were considered relevant, in general, with the most relevant results appearing well ranked very often. Finally, the improvement in the ranking of relevant search results over time was clearly noticed, especially by the developers of experiment A. That means that developers acknowledged the evolution of search weights as they adapted to their behaviour.

The questionnaire also asked developers about what they liked most in the application. Many developers said that the user interface as one of the best things, because it was simple, intuitive and easily accessible. Also, the search was reported as being fast and useful to quickly switch between source code artifacts. Other aspects mentioned as positive were the innovative concept behind the application, the improvement of the search results

Table 5.3: Questionnaire results for the context-based search, including the mean and the standard deviation, per group of developers.

		A		**B**		**A+B**	
Question	**Scale**	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
How would you rate the utility of the search functionality?	Very Low (1) - (5) Very High	4.20	± 0.51	4.70	± 0.94	4.53	± 1.36
How would you rate the usability of the search functionality?	Very Poor (1) - (5) Very Good	4.00	± 0.63	4.30	± 1.09	4.20	± 1.27
How would you rate the impact of the search functionality in your productivity?	Very Low (1) - (5) Very High	3.60	± 0.76	4.10	± 0.40	3.93	± 0.59
How would you rate the overall relevance of search results?	Very Irrelevant (1) - (5) Very Relevant	4.20	± 0.51	4.10	± 0.70	4.13	± 0.82
How often did relevant search results appear in search results?	Very Rarely (1) - (5) Very Often	4.80	± 0.51	4.90	± 0.40	4.87	± 0.63
How often did relevant search results appear well ranked in search results?	Very Rarely (1) - (5) Very Often	4.60	± 0.76	4.30	± 0.94	4.40	± 1.31
How would you rate the improvement in ranking of relevant search results over time?	Very Low (1) - (5) Very High	4.80	± 0.51	3.80	± 1.29	4.13	± 1.79

over time and the coherence between the application interface and the Eclipse interface. On the other hand, one of the things they liked the least was about the confusion created by artifacts with similar names but stored in different projects, which could be minimized by improving the way search results are presented to the developer. The developers gave interesting suggestions for improving the application. Some of them suggested that the application could give recommendations of relevant knowledge before performing a search, which could actually avoid the need to perform the search. Also, they would like to extend the context-based search to other elements and different types of files. Other suggestions were about the possibility of using filters in the search query, to allow filtering search results by package or type, and the use of auto-complete, to help building the search query. Finally, all developers said that would like to continue using the application in the future.

5.1.2 Context-Based Recommendation

In relation to the context-based recommendation, the objective of the experiment was to show evidence that our approach would help developers find relevant knowledge faster and efficiently. More specifically, we wanted to evaluate if the recommendations could be used to avoid the need to perform a search or browse the source code structure in order to find the needed elements. We also wanted to assess that the context of the developer could be used to identify the most relevant source code elements for the developer and that it would have a positive impact on the ranking of these elements. We have collected both quantitative and qualitative results from the experiment, that are discussed in the following sections. The results analyzed refer to the recommendation feature only, although being occasionally compared with the search results, for the sake of comprehension.

Quantitative Results

An immediate conclusion we could draw from a preliminary analysis of the results, both quantitative and qualitative, is that the recommendation functionality was somehow controversial, in the sense that some developers clearly found it useful and used it a lot, while others did not find it relevant and almost ignored it. This fact may have several explanations, but none could be directly drawn from the results we obtained from the experiment. We believe that one of two things may have happened, one is circumstantial and the other is behavioral. The circumstantial reason has to do with the specific characteristics of the tasks, or work environment, of developers, which may not be ideal for using a recommendation mechanism. For instance, if the work of the developer is spread among too many artifacts, with little relation to each other, the recommendation feature tend to be less useful. The behavioral reason arise from the fact that developers are not used to this kind of functionality, it is something they do not expect to see in the Integrated Development Environment (IDE) and those who are less open to new experiences tend to ignore it and continue their work.

Because there was a big difference between the developers that really used the recommendation feature and those who ignored it, we decided to analyze the quantitative results in two groups, one comprising all the 10 developers, called group C, and other comprising only the 5 developers who have used the recommendation more intensively, called group D. The later represents half of the developers and about 86% of the total number of recommendation results selected. These developers were also the most active, representing about 76% of all results selected, including search and recommendation.

As shown in table 5.4, developers selected a total of 214 recommendation results, from which 183 have been selected by developers of group D. About 71% of all the selected

Table 5.4: Number of selected recommendations, per interface and group of developers.

	C		D	
Search View	82	(38.3%)	79	(43.2%)
Search Window	70	(32.7%)	50	(27.3%)
Recommendation View (List)	46	(21.5%)	40	(21.9%)
Recommendation View (Cloud)	2	(0.9%)	2	(0.9%)
Recommendation Window (List)	7	(3.3%)	6	(3.3%)
Recommendation Window (Cloud)	7	(3.3%)	6	(3.3%)
Total	**214**	**(100%)**	**183**	**(100%)**

Table 5.5: The mean and confidence interval for the rankings of the selected recommendations, per component and group of developers.

	C		D	
	$\overline{x}$	**CI (95%)**	$\overline{x}$	**CI (95%)**
Interest Ranking (ri)	8.63	± 1.21	9.46	± 1.36
Time Ranking (rt)	7.26	± 1.05	7.98	± 1.18
Structural Ranking (rs)	6.14	± 0.83	6.45	± 0.91
Lexical Ranking (rl)	6.66	± 0.85	6.47	± 0.84
Final Ranking (rf)	5.07	± 0.62	5.40	± 0.71

results came from the search interface, which means that the developer had the intention to search for a specific source code artifact, but the desired artifact was recommended even before performing the search. The remaining recommendation results were selected using the recommendation view, but almost only from the interface that provides a list of results, which means the *code cloud* did not work out in practice and should be reconsidered. When we take into account all the results selected, both from search and recommendation, the recommendation results represent about 39% of all selected results, which rises to 43% for developers of group D. This means that almost half of the times, the need of the developer was satisfied by a recommendation. These numbers are a good indication that our recommendation approach is reducing the effort of the developer on finding relevant knowledge, avoiding the need to explore the source code structure or perform a search to find the desired artifacts.

The average final ranking of the selected recommendation results was 5.07, with an slightly superior ranking of 5.40 for developers of group D, see table 5.5. The average rankings of the individual components show how the result would be ranked if only that component was considered, and reflect how each component are influencing the final ranking. For group C, there is not a significant difference in the average rankings of the four components, with a small tendency for better structural and time rankings. When we analyze the average rankings of the developers in group D, there is a more significant difference between the retrieval (r_i and r_t) and the context (r_s and r_l) components, with the context components obtaining better rankings. These numbers indicate that the use of context is having a positive impact in the ranking of recommendation results, which would be ranked lower if only the retrieval components were used.

The average final recommendation weights denote a tendency to favor the time and context components compared to the interest component, as shown by the weighted mean of the final weights, presented in table 5.6. This tendency is more clear in the developers of group D, who have used the recommendation more frequently. As expected, the context

Table 5.6: The weighted mean for the final weights of the context-based recommendation, per component and group of developers.

	C		D	
	$\overline{x}$	σ	$\overline{x}$	σ
Interest Weight (w_i)	0.197	± 0.043	0.187	± 0.039
Time Weight (w_t)	0.280	± 0.075	0.282	± 0.081
Structural Weight (w_s)	0.242	± 0.109	0.242	± 0.118
Lexical Weight (w_l)	0.277	± 0.0716	0.286	± 0.0740

components gain more relevance over time, as they have a positive impact in the ranking of the selected recommendation results. On the other hand, we see that the time component plays a more important role, than the interest component, in the retrieval process.

Qualitative Results

By the end of the experiment, developers were asked to fill a questionnaire. The objective of this questionnaire was to perceive their opinion on the utility and quality of recommendation functionality. We also wanted to know what they liked the most, and the least, and what suggestions they would give to improve the prototype. The questionnaire was anonymous, so we could not split the analysis of the results the same way we did when analyzing the quantitative results. This way, we opted to split the developers into a group C, comprising all the 10 developers, and a group comprising the 5 developers which gave the best scores to the recommendation functionality, called group E.

As shown in table 5.7, when we consider the opinion of all the developers, the results show an average score for all the questions and a high standard deviation, which reflects the divergence of opinion among the developers. These results are coherent with those obtained in the quantitative analysis, as roughly half of the developers considered the recommendation functionality almost unhelpful while the others found it very useful. The results of group E show much better results, especially concerning the utility, usability and impact on productivity. The standard deviation is also much smaller, revealing an higher convergence of opinions. The results of the questions related with the ranking of the recommendation results are less impressive, similar to those obtained for all the developers. Such results show that some work must be done to improve the precision of the algorithm.

The questionnaire also asked developers about the things they liked the most and the least, and which suggestions they would give to improve the system. Most of the developers liked the idea of having a list of recommendations that could help predict their immediate needs. They pointed out that recommendations worked well as a quick jump list and could replace some of the interfaces for exploring the source code offered by the IDE. Some of them considered it more useful when they were working in a small context, as the recommendations were more accurate.

The problems pointed out about recommendation were essentially related with the user interface and the accuracy of the recommendations. According to some developers, the user interface uses too much of the space available in the IDE. They suggested that the recommendation interface should be completely merged with the search interface. The *code cloud* was considered an interesting interface but not useful. With respect to accuracy, some of the developers referred that occasionally the system would not suggest any relevant artifacts and, as expected, they would have to use the search. After all, 70% of the developers said they would like to continue using the recommendation functionality.

Table 5.7: Questionnaire results for the context-based recommendation, including the mean and the standard deviation, per group of developers.

		C		E	
Question	**Scale**	$\overline{x}$	σ	$\overline{x}$	σ
How would you rate the utility of the recommendation functionality?	Very Low (1) - (5) Very High	3.60	± 2.06	4.60	± 0.76
How would you rate the usability of the recommendation functionality?	Very Poor (1) - (5) Very Good	3.50	± 1.36	4.20	± 0.51
How would you rate the impact of the recommendation functionality in your productivity?	Very Low (1) - (5) Very High	2.80	± 2.82	4.00	± 0.63
How would you rate the overall relevance of recommendation results?	Very Irrelevant (1) - (5) Very Relevant	3.20	± 1.29	3.80	± 0.51
How often did relevant results appear in recommendation results?	Very Rarely (1) - (5) Very Often	3.30	± 1.09	3.60	± 0.76
How often did relevant results appear well ranked in recommendation results?	Very Rarely (1) - (5) Very Often	3.00	± 1.18	3.20	± 0.51
How would you rate the improvement in ranking of relevant recommendation results over time?	Very Low (1) - (5) Very High	3.10	± 1.37	3.40	± 0.76

5.2 Final Study

The final study was conducted after the analysis of the information collected during the preliminary study. This information was very important to evaluate our initial approach and detect issues that should be corrected. After applying the necessary changes to the prototype, which were already discussed in the previous section, we wanted to conduct an experiment similar to that of the preliminary study, but this time with a higher number of developers. This way, we have publicized the prototype within our university and in some communities. The prototype was installed and activated by 35 developers. Among these developers, 9 never uploaded usage data and 5 uploaded usage data that did not show any use of the search and recommendation features. Therefore, the results analyzed in this study were collected from 21 developers. Three of these developers have used the prototype in two different workspaces, which are referred as workspace A and B, for each developer. The number of days using the prototype and the knowledge base sizes for each developer are presented in table 5.8. The developers used the prototype during an average of 38 days, working with knowledge bases having an average of 3,496 structural elements, 9,370 structural relations, 679 lexical elements and 13,077 lexical relations. As in the preliminary study, we have collected statistical information about the context-based search and recommendation processes, but this time we analyze this information in more detail. In this study, we have also collected information about the context modeling process. The information collected is described and discussed in the following sections.

5.2.1 Context Model

Concerning the context modeling mechanism, we have collected statistical data about new elements added to the context model and how they were related with the elements that were already in the context model. This information would allow us to better understand how the source code elements manipulated by the developer are related with each other and how this could be used to improve the context modeling and transition processes. We have analyzed a total of 48,044 elements added to the context model, and have verified if they were related within a distance of 3 relations with the top 15 elements with higher interest, of both the structural and the lexical contexts. About 88% of the elements were structurally related with at least one structural element, within an average distance of 2.3 relations, and about 86% were lexically related with at least one lexical element, within an average distance of 2.0 relations. These numbers show that most of the source code elements accessed by developers were related, both structurally and lexically, with at least one of the elements manipulated before, within a distance of about two relations.

In table 5.9 we present the average number of structural and lexical elements in the context model, as well as the average number of elements that were related and unrelated with the added element. The lexical related elements have an higher average due to the fact that a source code element name typically comprises more than one term, and each match between one of these terms and the terms in the lexical context was considered. The results show that the source code elements added to the context model were structurally related with an average of almost 40% of the elements that were already in the context model. The results of the lexical elements are even more expressive. This reinforces the idea that the source code artifacts manipulated by developers are highly related. We have also analyzed the types of relations that are more common between the added elements and the existing elements. The percentage of times each relation appeared is shown in table 5.10. The composition and behavior relations are by far the most common, as expected.

With respect to the context transition process, we have collected statistical information about 109 context transitions, presented in table 5.11. All the context transitions led to

Table 5.8: The number of days of usage and average knowledge base sizes, per developer.

		Structural Ontology		Lexical Ontology	
Developer	**Days**	**Elements**	**Relations**	**Elements**	**Relations**
#1A	22	7,213	21,394	1,034	28,089
#1B	19	6,479	18,333	987	25,264
#2	32	12,089	30,983	980	35,186
#3A	21	5,227	13,737	761	18,094
#3B	79	7,581	21,788	1,531	39,058
#4	72	911	1,666	408	3,298
#5	2	743	1,509	317	3,108
#6	36	1,527	3,200	509	6,279
#7	20	14,690	46,666	1,753	41,845
#8	47	397	668	389	2,048
#9	68	2,697	5,603	798	11,839
#10A	10	892	1326	314	4269
#10B	28	178	274	163	803
#12	15	12,106	32,113	1,761	59,004
#15	102	949	2,543	354	3,994
#17	12	2,420	4,529	1,023	9,122
#19	30	265	449	489	2,402
#21	82	3,556	10,245	574	12,981
#24	18	139	282	128	954
#27	27	1,845	4,632	335	6,525
#28	23	3,463	10,169	1,258	11,549
#29	31	1,262	2,051	374	4,156
#30	6	252	329	156	1,048
#31	1	315	710	256	1,452
$\bar{x}$	**38**	**3,496**	**9,370**	**679**	**13,077**

Table 5.9: The average number of structural and lexical elements in the context model.

Average Structural Elements	11.69
Average Structural Related Elements	4.64
Average Structural Unrelated Elements	5.05
Average Lexical Elements	14.26
Average Lexical Related Elements	22.60
Average Lexical Unrelated Elements	3.82

the creation of a new context. We could conclude that a transition to a previous context is something very uncommon, but the problem may also reside in the rules we have defined for switching to an existing context. At first, the similarity between the two sets of elements was computed using the Jaccard index (Jaccard, 1901), also known as the Jaccard similarity coefficient, which is a statistical measure used for comparing the similarity between sample sets. When the similarity between the two sets was greater than a threshold of 0.5, the existing context would be activated. After some time, we verified

Table 5.10: The percentage of times each relation appeared in the relations between added and existing context elements.

Relation	Percentage
extensionOf	8.57%
implementationOf	1.38%
attributeOf	36.62%
methodOf	90.73%
parameterOf	10.37%
returnOf	3.07%
calledBy	49.75%
usedBy	31.19%

Table 5.11: The statistical information collected about the context transition process.

Context Transition	Percentage
New Context	100%
Existing Context	0%
Hard Transition	73%
Soft Transition	27%

that the context transitions to existing context models were not happening and realized that using the Jaccard metric would not be adequate for our purpose. The two sets being compared were very often disproportionate, because the number of transitive elements tends to be small compared to the number of elements in a context model. Hence, the threshold was never reached and none of the existing contexts would be activated.

Being aware of this problem, we have refined the conditions to activate an existing context, focusing only in the transition elements. Because the transition elements are more relevant for the context transition process than the others, we may assume that if an existing context contains all the transition elements, then this context model is a good candidate to be activated (see section 3.2.3). Despite this change, there were no context transitions to an existing context. This way, we tend to believe that the transitions to an existing context are rare, but the process must be better studied in order to draw stronger conclusions. With respect to the transitive elements, we could conclude that context transitions are more often caused by reaching the hard transitive elements threshold, than by reaching the soft transitive elements threshold. This was expected, because the hard transitive elements are more relevant to the process and therefore have a lower threshold. But, it also shows that the soft transitive elements have their role in detecting context transitions.

Finally, we asked the developers to evaluate how each context transition detected by the system could be identified as a change in their focus of attention. They were presented with the structural elements that were in the context model before the transition and the elements that were used to detect the transition, both hard and soft transitive. They were asked to rate how the context transition would be related with a change in their focus of attention, in a scale from 1 (Poorly Related) to 5 (Highly Related). The average score for the 55 context transitions evaluated was 3.00, with a confidence interval of ± 0.33, for a confidence level of 95%. The average score obtained is not conclusive, but is encouraging, at least. One of the problems we have faced is that developers have some difficulties understanding the concepts of context transition and focus of attention, which

can have lead to misjudgment in the evaluation process. When we defined the context transition process, we were seeking to endow our approach of a mechanism to deal with situations when the developer changes the focus of attention to different parts of the source code. Therefore, our definition of context transition is very practical and focused on improving the retrieval mechanisms of our approach. But, when we ask developers about context transitions and changes in their focus of attention, they may create very different interpretations for these concepts, due to their ambiguity. Because of this, they may end up giving an average classification to a context transition, just because they do not understand what is being asked.

Also, we have asked developers to evaluate the context transitions some time after these have occurred, because we did not want to interrupt their work every time a context transition was detected. This could also pose a problem to the evaluation process, because the source code is constantly evolving and as time passes a context transition becomes more difficult to evaluate, or even to understand. This could be overcome by asking the developers to evaluate a context transition in the moment it occurs, which inevitably would lead to interruptions in their work. Another approach could be asking the developers to explicitly indicate a context transition, but this would require an extra effort from them and would also be affected by the interpretation problems we have referred before.

5.2.2 Context-Based Search

Concerning the context-based search, our evaluation was similar to that described in the preliminary study. This way, we wanted to find evidence that the use of the context model was having a positive impact on the ranking of the search results. The quantitative and qualitative results collected are described in detail in the following sections.

Quantitative Results

During this study, the developers selected a total of 1120 search results. Among the searches with selected search results, the search queries used by the developers had an average size of 6.36 ± 0.26 characters. The reduced size of the search queries may be indicative that the use of context reduces the need of using larger search queries. The incremental search mechanism that was used in the search interface may have also contributed to shorter queries, because the search results were being displayed automatically as the query was being written. The searches returned an average of 12.22 ± 0.61 results, taking into account the limit of 30 results presented to the developer. The search process was performed on an average of $213.65 \pm 20.87ms$, which is within our objective of around $1s$. These values and respective confidence intervals were computed for a confidence level of 95%. As shown in table 5.12, the search results selected had, in average, a final ranking of 2.40, a retrieval ranking of 8.03, a structural ranking of 2.31 and a lexical ranking of 5.34. These values are marginally better and consistent with the results obtained in the preliminary study, showing evidence that the context components had a positive influence in the final ranking of the search results. We have also stored the ranking of the search results if only one of the context components was combined with the retrieval component, so that they could be evaluated separately (see table 5.12). The results show that, in average, the structural and retrieval components combined would result in a final ranking of 2.08, while the lexical component combined with the retrieval one would result in a final ranking of 4.78. These values confirm, once again, that the context components have a positive impact in the ranking of search results, even when used individually, especially the structural component. Another interesting conclusion we can draw is that search results would be slightly better ranked if the lexical component was ignored. Although this

Table 5.12: The mean and confidence interval for the rankings of the selected search results, per component.

	$\overline{x}$	CI (95%)
Retrieval Ranking (rr)	8.03	± 0.50
Structural Ranking (rs)	2.31	± 0.19
Lexical Ranking (rl)	5.34	± 0.42
Final Ranking (rf)	2.40	± 0.16
Retrieval/Structural Ranking	2.08	± 0.16
Retrieval/Lexical Ranking	4.78	± 0.37

Table 5.13: Comparison between the rankings of the individual components for the selected search results.

			Difference	
Comparison	**Count**	**Percentage**	$\overline{x}$	**CI (95%)**
Structural Ranking (rs) < Retrieval Ranking (rr)	801	71.5%	8.96	± 0.59
Structural Ranking (rs) = Retrieval Ranking (rr)	243	21.7%	—	—
Structural Ranking (rs) > Retrieval Ranking (rr)	76	6.8%	3.75	± 0.99
Lexical Ranking (rl) < Retrieval Ranking (rr)	693	61.9%	7.44	± 0.57
Lexical Ranking (rl) = Retrieval Ranking (rr)	272	24.3%	—	—
Lexical Ranking (rl) > Retrieval Ranking (rr)	155	13.8%	6.36	± 1.01
Final Ranking (rf) < Retrieval Ranking (rr)	665	59.4%	10.02	± 0.68
Final Ranking (rf) = Retrieval Ranking (rr)	329	30.4%	—	—
Final Ranking (rf) > Retrieval Ranking (rr)	126	11.3%	2.81	± 0.54

is true, we believe that the lexical component should not be neglected, because it plays an important role in those cases where the search result does not get a structural score.

Besides analyzing the average rankings for each component, we have also investigated if the search results were effectively getting better ranked by using the context model in the ranking process. As presented in table 5.13, in comparison with the retrieval component, the search results were better ranked in almost 60% of the times, being worse ranked in only 11% of the times. The average rankings difference when the search results were better ranked was 10.02, being only 2.81 when they were worse ranked. Although there were worse ranked results in 11% of the times, the difference in the ranking of these results was much smaller when compared to the improvement in the rankings of the better ranked results. The differences between the final and the retrieval rankings were clearly influenced by the context components, with the structural component getting better rankings in about 71% of the times and the lexical component in about 62% of the times.

The evidence that the context of the developer is having a positive impact in the ranking of search results is reinforced when we analyze the average rankings for each developer, which are presented in table 5.14. The average ranking of the structural component is consistently lower across almost all the developers, especially with those who have selected more results (see for instance developers #1, #4, #6, #9 and #21). Regarding the lexical component, the results are not so clear, but it still gets better average rankings with most of the developers with higher number of selected results (see for instance developers #4, #6, #9 and #21).

The weighted mean of the final search weights, according to the number of selected search results per developer, is presented in table 5.15. The differences between the three

Table 5.14: The number of selected results, average rankings and final weights for the context-based search, per developer.

		Rankings				Weights		
Developer	**Results**	r_r	r_s	r_l	r_f	w_r	w_s	w_l
#1A	59	9.46	4.02	11.04	3.00	0.343	0.395	0.262
#1B	35	10.46	4.46	11.74	3.17	0.337	0.372	0.291
#2	30	20.50	2.44	8.45	2.60	0.255	0.392	0.352
#3A	42	1.93	1.31	2.14	1.60	0.341	0.337	0.315
#3B	32	6.94	2.00	4.17	1.84	0.321	0.349	0.331
#4	427	9.79	1.88	4.48	2.21	0.00	0.745	0.255
#5	5	2.80	2.20	2.60	2.20	0.341	0.329	0.330
#6	114	7.65	1.43	5.26	2.17	0.217	0.446	0.338
#7	9	7.44	7.25	12.22	8.11	0.362	0.352	0.286
#8	5	3.80	1.50	1.00	2.20	0.338	0.326	0.332
#9	100	10.25	4.47	7.54	3.29	0.270	0.387	0.309
#10A	6	2.33	1.33	1.67	2.33	0.328	0.339	0.333
#10B	1	3.00	1.00	7.00	3.00	0.341	0.333	0.325
#12	10	8.60	1.75	10.00	2.70	0.340	0.340	0.320
#15	164	2.27	2.05	2.95	2.27	0.271	0.431	0.298
#17	0	—	—	—	—	—	—	—
#19	7	2.71	1.17	1.25	1.14	0.335	0.334	0.331
#21	30	10.90	2.43	5.92	2.20	0.273	0.382	0.344
#24	9	1.33	1.33	1.75	1.33	0.342	0.332	0.325
#27	18	5.78	1.67	2.14	2.00	0.336	0.330	0.334
#28	4	1.50	1.00	1.00	1.00	0.338	0.331	0.331
#29	0	—	—	—	—	—	—	—
#30	8	1.38	1.86	1.60	1.50	0.343	0.323	0.333
#31	5	4.00	1.00	4.00	4.00	0.340	0.330	0.330

weights are consistent with those obtained in the preliminary study, with an average retrieval weight of 0.177, an average structural weight of 0.529, and an average lexical weight of 0.291. It is clear that the context components are being favored in relation to the retrieval component. This is more evident when we look at the final weights, shown in table 5.14, of the developers with a higher number of selected results (see for instance developers #4, #6, #9 and #15). The evolution of the weights of these developers over time is shown in figure 5.1. A fact worth mentioning is that in the case of developer #4, who have selected, by far, the highest number of search results, the weight of the retrieval component was reduced to zero. This means that the retrieval process was used only for retrieving the search results, but was not contributing for their ranking, which was computed using only the context components.

Qualitative Results

By the end of the experiment, we requested the developers to fill an anonymous questionnaire. The objective was to collect their opinion about the context-based search feature. We collected feedback results from 12 developers that used the prototype, which are presented in table 5.16. The average classification for all the questions was very positive,

Table 5.15: The weighted mean for the final weights of the context-based search, per component.

	$\overline{x}$	σ
Retrieval Weight (w_r)	0.177	± 0.143
Structural Weight (w_s)	0.529	± 0.172
Lexical Weight (w_l)	0.291	± 0.034

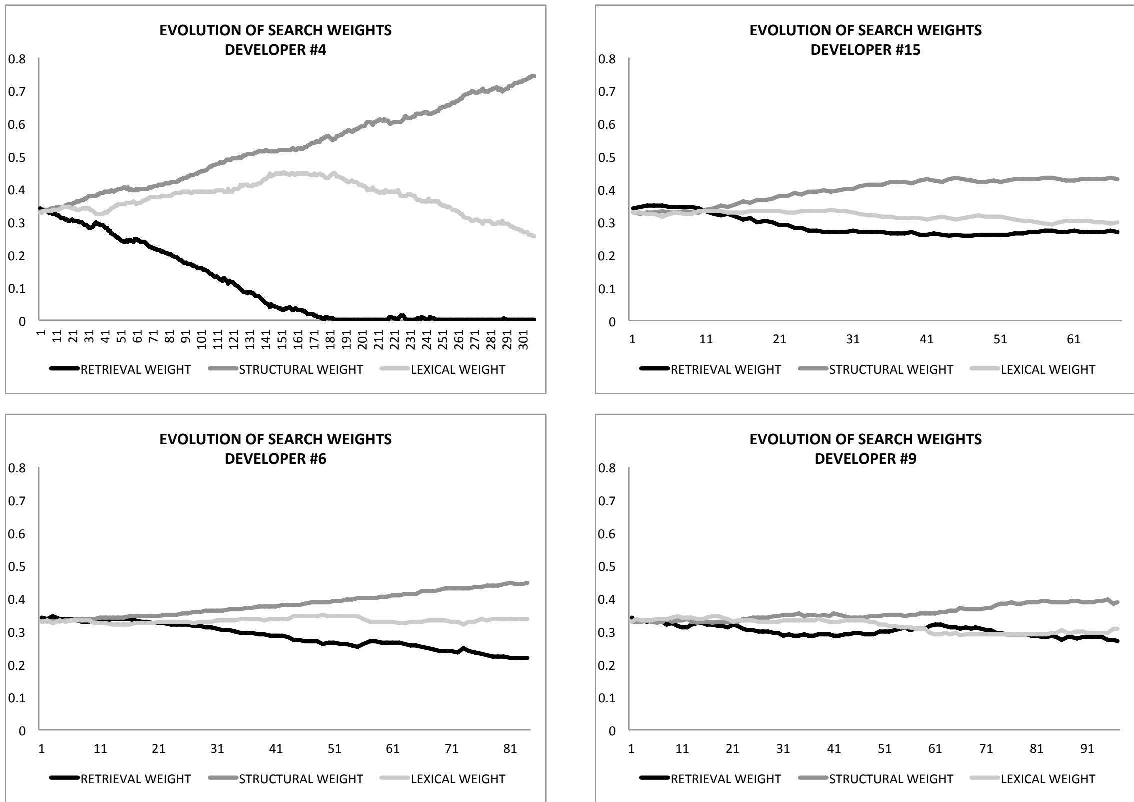

Figure 5.1: Evolution of the search weights for developers #4, #15, #6 and #9, where the x-axis represents each weights update and the y-axis represents the value of the weights.

especially for the questions related to utility and usability. The questions related with the improvement in their productivity and in the ranking of search results over time, obtained the less expressive classifications and the higher variance.

With respect to the things they liked the most, most of the developers referred that the search feature was very fast, efficient and simple to use, saving time and reducing the effort needed to search for desired elements. They highlighted the fact that it was very useful to jump from class to class, and method to method, avoiding the need to navigate the source code structure. The search results were considered very accurate and better ranked than traditional search.

Some of the things that developers liked the least where related with implementation problems, such as synchronization issues in the incremental search feature and the resource consumption of the prototype. The limitation to search only the identifiers of the source code elements was noted by several developers. Also, it was referred that when changing contexts very fast, the relevant search results took some time to appear well ranked. Another limitation that was noted was the inability to find inner classes, which are classes that are defined inside other classes.

Taking into account the limitations identified, the developers gave several suggestions for improving the prototype. They wanted the search to be performed in the entire content of the source code elements, for instance including variable names and comments. Concerning the user interface, they would like to be able to expand search results, see the search results organized in clusters and avoid the search results that are already opened in that moment.

5.2.3 Context-Based Recommendation

As in the preliminary study, the context-based recommendation evaluation aimed to collect evidence that the recommendations could be used to avoid the need of performing a search or browsing the source code structure to find the needed elements. In the following sections, we present the quantitative and qualitative results that were collected. Although these results refer to the recommendation process only, they are occasionally compared with the search results, for the sake of comprehension.

Quantitative Results

At first, we wanted to evaluate the capacity of the system in predicting the source code elements that the developers would need in the near future, so that these elements could be pro-actively recommended to them. Also, we wanted to discover what value of N should be used to achieve the best results (see section 3.4.1). This evaluation was performed in the background, by verifying if the source code elements being opened, or accessed, for the first time were already being recommended by the system. This way, we were able to evaluate our approach using the behavior of the developers during their work, without requiring them to use our recommendations. We have implemented a mechanism to store the top 30 recommendations generated by the system with a random value of N (between 1 and 10). For each source code element opened, or accessed, for the first time, we have verified if that element was being recommended by the system in that moment.

In table 5.17, we present the results obtained per each value of N, along with the total number of elements found in recommendations, the percentage of elements found, and the average final rankings. In average, considering all values of N, 41.14% of the source code elements opened, or accessed, for the first time were already being recommended by the system. The best results were achieved with a value of 2 for N, with which the system has been able to predict the developer needs in 52.98% of the times. With a

Table 5.16: The questionnaire results for the context-based search, including mean and standard deviation.

Question	**Scale**	$\overline{x}$	σ
How would you rate the utility of the search functionality?	Very Low (1) - (5) Very High	4.75	± 0.92
How would you rate the usability of the search functionality?	Very Poor (1) - (5) Very Good	4.75	± 0.92
How would you rate the impact of the search functionality in your productivity?	Very Low (1) - (5) Very High	4.33	± 1.19
How would you rate the overall relevance of search results?	Very Irrelevant (1) - (5) Very Relevant	4.42	± 1.24
How often did relevant search results appear in search results?	Very Rarely (1) - (5) Very Often	4.50	± 1.22
How often did relevant search results appear well ranked in search results?	Very Rarely (1) - (5) Very Often	4.42	± 1.24
How would you rate the improvement in ranking of relevant search results over time?	Very Low (1) - (5) Very High	4.08	± 1.42

Table 5.17: The number, percentage and average rankings of newly accessed source code elements found in recommendations, per value of N.

			Rankings				
N	**Count**	**Percentage**	r_f	r_i	r_t	r_s	r_l
1	1340/3338	40.14%	6.52	7.67	6.56	7.10	5.81
2	1743/3290	52.98%	10.25	10.26	11.10	9.79	8.61
3	1678/3254	51.57%	12.64	11.55	14.86	11.54	10.01
4	1565/3251	48.14%	14.12	13.37	17.11	12.50	10.49
5	1443/3255	44.33%	14.84	13.52	18.83	12.34	11.14
6	1315/3315	39.67%	15.26	13.65	19.79	12.46	11.21
7	1144/3174	36.04%	15.36	13.92	20.18	12.10	11.39
8	1043/3113	33.51%	15.36	13.94	20.35	12.39	11.59
9	1038/3179	32.65%	14.94	13.51	20.54	11.76	11.19
10	1004/3195	31.42%	15.27	13.95	20.35	12.01	11.17
Total	**13313/32364**	**41.14%**	**13.21**	**12.35**	**16.46**	**11.32**	**10.14**

value of 3 for N, the percentage of predicted elements was 51.57%, which is also very close to the best percentage obtained for a value of 2. As expected, the results also show that very lower values of N tend to have worse values, as the number of source code elements used to retrieve the recommendations is not enough to reach the desired element. The higher values of N also have worse results, which can be explained by the fact that when we increase the number of source code elements in the retrieval process, the recommendations became more dispersed and the probability of finding what the developer needs decreases. We believe that these results are very interesting and show that the context of developers has much to say about their immediate needs. Although the rankings can still be improved, the results also show a slight tendency for better rankings obtained with the context components, which reveals a positive impact of the context model in the final ranking.

We have also collected information about the recommendations selected by the developers. Among the list of recommendations with selected recommendations, the average number of recommendations presented to the developer was 25.34 ± 0.70, taking into account the maximum of 30 recommendations that could be presented. The recommendation process was performed on an average of $292.86 \pm 47.02ms$, which is within our objective of around $1s$. These values and respective confidence intervals were computed for a confidence level of 95%. As shown in table 5.18, a total of 379 recommendations were selected. About 93% were selected from the recommendations integrated in the search interfaces, while the recommendation only interfaces were almost ignored. This difference is even more expressive than that obtained in the preliminary study. We believe this can be explained by the fact that an interface that provides pro-active recommendations and also allows to perform a search delivers more value to the developer. This may also indicate that the developer had the intention to search for a specific source code element, but the desired element was being recommended even before performing the search. When we take into account all the source code elements selected, both from search and recommendation, the recommendations represent about 34% of all selected elements, which is also consistent with the results of the preliminary study. That means that in 34% of the times in which the developer used our prototype to reach a desired source code element, the need of the developer was satisfied by a recommendation.

Table 5.18: Number of selected recommendations, per interface.

Search View	174	(45.9%)
Search Window	180	(47.5%)
Recommendation View (List)	11	(2.9%)
Recommendation View (Cloud)	13	(3.4%)
Recommendation Window (List)	0	(0%)
Recommendation Window (Cloud)	1	(0.3%)
Total	**379**	**(100%)**

Table 5.19: The mean and confidence interval for the rankings of the selected recommendations, per component.

	$\overline{x}$	**CI (95%)**
Interest Ranking (ri)	9.08	± 0.77
Time Ranking (rt)	8.96	± 0.80
Structural Ranking (rs)	8.03	± 0.76
Lexical Ranking (rl)	7.41	± 0.68
Final Ranking (rf)	5.40	± 0.55

As shown in table 5.19, the average final ranking of the selected recommendations was 5.40. Although it may still be subject to improvements, we consider that this is a good precision for this type of recommendation system. The average rankings of the individual components were 9.08 for the interest component, 8.96 for the time component, 8.03 for the structural component and 7.41 for the lexical component. These rankings do not show a significant difference between the four components. Although the context components have slightly better average rankings, the difference is lower than the difference obtained in the preliminary study. Interestingly, the lexical component achieves better rankings than the structural one, which depicts that the lexical relations between source code elements are contributing to improve the precision of the recommendations. When we analyze the average rankings for each developer, which are presented in table 5.20, we verify that there is not a convergence about which components may be more relevant. For instance, among the developers who have selected more recommendation results, there are cases where the context components had better average rankings (#4, #7 and #8) and others where the interest or the time components obtained better average rankings (#2, #6 and #27).

The weighted means for the final recommendation weights are presented in table 5.21. In accordance with the average rankings, the weights do not show evidence of a significant difference between the four components, although there is a slight favoring of the context components. Also, for the weights to converge it is necessary to select a significant number of recommendations, and when we consider the weights of the developers who have selected a higher number of recommendations, as shown in table 5.20, the differences become more explicit. For instance, the weights of developer #4 show a clear predominance of the context components, the weights of developer #10A were favoring the structural and the interest components, and the weights of developer #21 and #7 were tending to favor the structural and lexical components. The evolution of the recommendation weights for these developers is shown in figure 5.2.

Table 5.20: The number of selected results, average rankings and final weights for the context-based recommendation, per developer.

		Rankings					**Weights**			
Developer	**Results**	r_i	r_t	r_s	r_l	r_f	w_i	w_t	w_s	w_l
#1A	16	13.94	14.25	10.13	10.25	7.00	0.248	0.225	0.282	0.245
#1B	6	11.67	4.00	21.17	7.83	4.83	0.240	0.268	0.233	0.259
#2	10	6.50	5.40	7.70	9.10	3.50	0.250	0.253	0.244	0.254
#3A	0	—	—	—	—	—	—	—	—	—
#3B	0	—	—	—	—	—	—	—	—	—
#4	128	10.10	8.05	7.09	4.84	3.64	0.130	0.174	0.332	0.364
#5	8	7.00	11.50	10.88	9.88	12.13	0.285	0.230	0.235	0.250
#6	38	5.79	6.34	7.39	6.68	3.92	0.274	0.234	0.236	0.257
#7	16	15.38	20.38	5.53	10.15	9.25	0.239	0.185	0.305	0.271
#8	22	7.86	9.14	5.63	2.29	5.45	0.262	0.202	0.269	0.261
#9	1	2.00	16.00	6.00	18.00	2.00	0.252	0.249	0.251	0.248
#10A	63	9.06	9.30	8.90	11.31	9.37	0.318	0.265	0.343	0.074
#10B	2	2.00	2.00	11.50	3.00	1.50	0.254	0.249	0.247	0.251
#12	2	1.00	1.50	12.00	4.50	1.50	0.252	0.252	0.245	0.251
#15	2	5.50	9.50	4.00	6.00	4.00	0.250	0.246	0.254	0.250
#17	4	2.25	2.00	2.50	2.50	2.50	0.250	0.258	0.244	0.248
#19	0	—	—	—	—	—	—	—	—	—
#21	33	10.64	12.76	10.72	8.58	5.42	0.246	0.201	0.262	0.291
#24	3	12.33	13.67	12.33	3.33	4.33	0.251	0.241	0.251	0.258
#27	21	3.71	3.05	6.00	10.31	2.86	0.261	0.245	0.261	0.232
#28	1	18.00	12.00	7.00	15.00	13.00	0.244	0.251	0.258	0.247
#29	1	7.00	11.00	13.00	4.00	4.00	0.251	0.248	0.246	0.254
#30	1	1.00	1.00	2.00	1.00	1.00	0.251	0.251	0.248	0.251
#31	1	4.00	12.00	8.00	4.00	5.00	0.253	0.244	0.250	0.253

Table 5.21: The weighted mean for the final weights of the context-based recommendation, per component.

	$\overline{x}$	σ
Interest Weight (w_i)	0.224	± 0.071
Time Weight (w_t)	0.214	± 0.035
Structural Weight (w_s)	0.297	± 0.041
Lexical Weight (w_l)	0.264	± 0.098

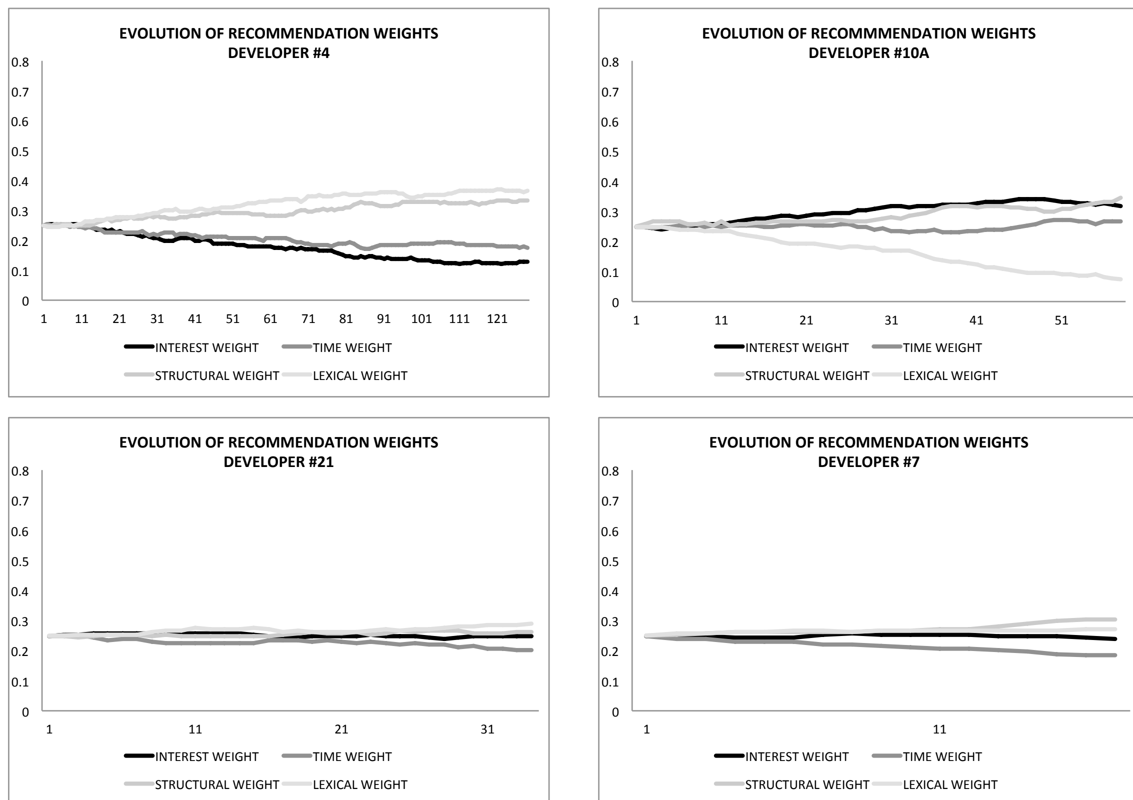

Figure 5.2: Evolution of the recommendation weights for developers #4, #10A, #6 and #21, where the x-axis represents each weights update and the y-axis represents the value of the weights.

Qualitative Results

The developers were requested to fill an anonymous questionnaire to give their opinion about the recommendations provided by the system. We collected feedback results from 15 developers that used the prototype, which are presented in table 5.22. Although the average classification for all the questions is positive, there are some differences that should be highlighted. The usability of the system was considered good, and the relevance of recommendations was rated positively. The utility of the recommendations and the learning mechanism were also rated above the average. From these results we also conclude that the raking of recommendations is something that can be improved. The utility and impact in the productivity obtained the lowest score among all the questions.

We also asked developers about the things they liked the most and the least, and which suggestions they would give to improve the system. With respect to what they liked, most of the developers said the recommendations were an easy, fast and useful way to jump into the desired source code elements. They said it was saving the time needed to investigate the source code structure, look inside source code files or even perform a search.

Concerning the things they liked the least in the recommendation feature, a few of the developers said that performing a search was sometimes more efficient than looking for the desired elements in the list recommendations. This was sometimes related with the need to improve the accuracy of the approach, so that the relevant recommendation appear better ranked. Some of the developers noted that most of the times the recommendations included elements that were currently opened and active, which should be avoided because the recommendation of such elements is irrelevant. Also, the *code cloud* was not considered useful by several developers.

Taking into account their analysis of the prototype, developers have proposed several suggestions. Most of the suggestions were related with the user interface. For instance, some of them wanted to be able to expand the recommendations and see more information about them. Because the recommendations were sometimes confused with search results, they suggested that they should be somehow differentiated. The capacity of the system in dealing with fast context transitions is something that could also be improved. They also suggested that recommendations could be used to improve the code-completion feature, the source code navigation and to highlight relevant methods directly in the source code.

5.3 Discussion

We presented the experiments we have performed to validate and evaluate our approach. These experiments were carried out in a real world scenario, with developers using our prototype during their daily work. Although these real world experiments pose several challenges and are difficult to implement at a large scale, we managed to run the experiment with a reasonable number of developers. We started with a *preliminary study*, involving a smaller group of developers, which allowed us to evaluate an initial version of our prototype. The feedback we have collected during this experiment was essential to identify a number of issues that had to be improved. A a good portion of these issues were related with the user interface, showing that usability was a crucial factor. We have also identified some issues in the various components of our approach, which were corrected before the *final study*.

With respect to the *context model*, we have collected statistical information about how the source code elements accessed by the developer are related between each other. The results showed that these elements are highly related, being structurally or lexically related with other elements already in the context model in more than 80% of the times. Which reinforces our assumption that the relations between an arbitrary source code element and the elements in the context model could be used to access the relevance of that element to the developer. Concerning the context transition mechanism, we asked developers to evaluate a set of context transitions, but concluded that it was hard to evaluate, due to the different interpretations that a context transition may have.

The *context-based search* process was evaluated by comparing the different components that contribute to the final ranking of a search result. We analyzed the search results that were selected by the developer, assuming that these results were considered more relevant than the remaining. From this analysis, we have concluded that the results were most of the times better ranked using our approach than they would if using only the retrieval model. In fact, they were better ranked in about 60% of the times, and worst ranked in only 11% of the times. This pattern was consistent in the two experiments and fairly regular among all the developers, showing evidence that the context model has a very positive impact in the ranking of the search results. This was also visible in the evolution of the weights used to balance the contribution of each component, with the learning mechanism tending to favor the context components in detriment of the retrieval component. Given the difference between the three components, the weight learning process played a central role in improving the ranking of the search results.

Regarding the *context-based recommendation* process, we evaluated the capacity of our approach in predicting the source code elements needed by the developer in the near future. This was done by analysing if the source code elements accessed by the developers, during their work, were already being recommended by the system. By taking into account the two context elements with higher interest and accessed more recently, our approach was able to predict the element being accessed in about 53% of the times, being ranked

Table 5.22: Questionnaire results for the context-based recommendation, including the mean and the standard deviation.

Question	**Scale**	$\overline{x}$	σ
How would you rate the utility of the recommendation functionality?	Very Low (1) - (5) Very High	3.80	± 1.58
How would you rate the usability of the recommendation functionality?	Very Poor (1) - (5) Very Good	4.33	± 1.29
How would you rate the impact of the recommendation functionality in your productivity?	Very Low (1) - (5) Very High	3.27	± 1.52
How would you rate the overall relevance of recommendations?	Very Irrelevant (1) - (5) Very Relevant	4.00	± 0.73
How often did relevant recommendations appear among all recommendations?	Very Rarely (1) - (5) Very Often	3.93	± 1.09
How often did relevant recommendations appear well ranked among all recommendations?	Very Rarely (1) - (5) Very Often	3.67	± 1.29
How would you rate the improvement in ranking of relevant recommendations over time?	Very Low (1) - (5) Very High	3.80	± 1.58

on average at position 11. We believe this is a clear evidence that the context model has a very important role on identifying the source code elements that may be of interest to the developer. When it comes to analysing the importance of the different components that contribute to the ranking of recommendations, the results are not as consistent as those obtained for the search process. As opposed to search, where we rely on the query of the developer to retrieve the desired source code elements, in recommendation we rely exclusively on the context model to retrieve and rank recommendations, which are only predicted to be relevant for the developer. Thus, the effectiveness of the recommendation process is highly dependent on the context of the developer. When we analyzed the recommendations that were selected by the developer, the results were not as evident as in search. The four components that contribute to the final ranking of recommendations had different results for different developers. This way, we were not able to conclude which components are more relevant to rank recommendations, as their importance change from developer to developer. This fact reinforces the importance of using a weight learning mechanism, allowing the system to adapt the contribution of these components to the different characteristics of each individual developer.

The qualitative assessment was performed using anonymous questionnaires, to collect the opinion of the developers in relation to the context-based search and recommendation features. The opinion about the search feature was very positive and consensual among the developers, being considered very fast, efficient and simple to use. The major limitation identified by the developers was related with the inability of searching the entire content of the source code elements, since the search was limited to the identifiers of such elements. With respect to the recommendation feature, the results were not so evident. Although it was also considered, by most of the developers, an easy, fast and useful way to jump into the desired source code elements, there is still a lot of space for improvements. The suggestions provided by the developers focus on improving the accuracy of the recommendations and the user interface.

5.4 Limitations

The results obtained in our experiments should be analyzed in light of some limitations. First, we chose not to conduct a laboratory study, because we believe that the ambiguity and complexity associated to the context of a developer could not be simulated is such an environment. This way, we conducted our experiments with developers in their work environment, so that our study could be as closest as possible of a real world scenario. Although we believe this was the best way of validating our approach, we had no control over the representativity of the source code base of each developer, or the tasks performed by these developers. We have collected additional information about the size and structure of the source code base used by each developer, which can be indicative but not completely representative of its complexity and scope. Concerning the tasks performed by the developers, we tried not to focus on individual tasks, thus we have no information related with the kind of tasks that were addressed. We do not know to what extent the source code base or the kind of tasks being addressed may be influencing the results that were obtained. Also, we have conducted the study with volunteers, this way we could not guarantee that the developers would be representative in terms of different aspects, such as experience, development methods, time constraints, domain, etc. Nevertheless, we tried to validate our approach with a diversified group of developers, from both industry and academia.

Considering the use of the search results and recommendations selected by the developers to access the impact of the context in the retrieval mechanisms, we are assuming

that these selected source code elements were relevant for the developers. Although it is a common assumption in this kind of evaluation, we can not assure that they were in fact relevant for the work of the developer. This assumption avoids the need to obtain explicit feedback from the developers, which would end up interrupting their work. The strategy used works like an implicit feedback mechanism, but may lead to wrong assumptions in some cases.

Finally, the prototype we have implemented is limited to the Java programming language and can be used only in the Eclipse IDE. Although these limitations must be taken into account, our choices were weighted taking into account the characteristics of the selected programming language and IDE. The Java programming language is one of the most used programming languages and we believe it is representative of the great majority of object oriented programming languages. The Eclipse IDE is also one of the most used IDEs for the Java programming language, being one of the most complete with respect to the features and tools available to the developers.

Chapter 6

Related Work

"Research is to see what everybody else has seen,
and to think what nobody else has thought."

Albert Szent-Györgyi

This chapter presents several works, in different areas, that are related with our work, either because they have similar objectives, or because they have applied techniques that are similar to those we have used. We categorize these works in four main sections, according to where they have been applied. First, we introduce works related with context awareness in software development, that is where our work is grounded and where a more detailed comparison is provided. Next, we describe some works related to software exploration, where search and recommendation have been largely applied to help exploring a software system. Following, several works in the area of software reuse are presented, which also make use of search and recommendation to foster reuse in software development. Finally, we present some of the works that have used the history of a software development project to provide guidance to software developers.

6.1 Context Awareness in Software Development

The increasing dimension and complexity of software systems, as well as the nature of the work of a software developer, led to a situation where developers need to handle several complex tasks, in a single day of work, and keep track of the different contexts associated to these tasks. As an attempt to overcome the problems associated to task complexity and task switching, several approaches have applied context to improve awareness and help recovering the mental state associated to a task. Most of these approaches use the interaction history of the developer to infer and model the context associated to a task. Some of them use the interaction history and navigation paths of different developers to identify which artifacts are more relevant in a specific task context. Our approach as been inspired by some of these works, in the way we also use the interaction history to model the context, or the focus of attention, of the developer in a specific moment, to aid search and exploration of the source code. Following, we describe some of the existing approaches and compare them with our work.

Kersten and Murphy (Kersten and Murphy, 2005, 2006) have developed a model for representing tasks and their context. They define a task as *"a usually assigned piece of work often to be finished within a certain time"*, which for a programmer includes bug fixes, feature additions and code base explorations. Also, they define a task context as *"the information - a graph of elements and relationships of program artifacts - that a programmer needs to know to complete that task"*. This task context is derived from an

interaction history, which comprises a sequence of interaction events representing operations performed on a software program's artifact. The interaction history representing the activity performed for a task is then processed and a graph is created, representing the task context. In this graph, nodes represent program's artifacts and edges represent the programming language reference relations. The task's interaction history is used to compute a weight for the elements in the task context, which is a real number representing the element's Degree of Interest (DOI) for the task. This approach to element weighting is loosely based on the model proposed by Card and Nation (Card and Nation, 2002), which have used a DOI model together with focus+context visualization techniques to create attention-reactive interfaces for hierarchical information.

They then use the information in a task context either to help focus the information displayed in the Integrated Development Environment (IDE), or to automate the retrieval of relevant information for completing a task. In order to support their investigation effort they developed Mylar, which integrates the task context model with the Eclipse[1]. The Mylar IDE integration allows programmers to work with task context in several commonly used parts of Eclipse, namely:

- DOI-based element decoration of Java, XML and files;
- DOI-based filtering in tree and list views;
- DOI-based ranking of elements and relationships in table views;
- DOI-based folding in the editor.

Furthermore, Mylar provides task context specific functionalities such as:

- Active Search View, which shows elements and relations of predicted interest;
- Active Test Suite, which creates and runs unit tests in the task context;
- Active Hierarchy, which shows the inheritance context of the task.

To validate their theory, they performed a longitudinal field study, which was representative of the real long-term tasks performed on large systems on industry, obtaining both quantitative and qualitative evidence that their approach can make programmers more productive (Kersten and Murphy, 2006). Their prototype has evolved to the Mylyn[2] project, and the task focused interface they developed is currently part of the Eclipse IDE.

We have also used a degree of interest model to represent the context of the developer in a specific moment, which is inspired by the model proposed by Kersten and Murphy. As in their work, we rely on the interactions of the developer in the IDE to build this model. However, we monitor only source code elements, such as classes, interfaces and methods, that are opened, activated, edited or closed by the developer. Based on the DOI of these elements, we compute a DOI for the structural relations that exist between them. The DOI we assign to each relation represents the relevance of that type of relation to the developer, and not the relevance of a specific relation between two source code elements. Also, we have extended this model with a lexical perspective, by assigning a DOI to the terms that comprise the identifiers of the source code elements that are represented in the context model of the developer.

Instead of using the context model to help focus the User Interface (UI) in the elements that are more relevant to the developer, we use this model to improve the retrieval of source

[1] `http://www.eclipse.org/` (August 2012)
[2] `http://www.eclipse.org/mylyn/` (August 2012)

code elements through search and recommendation. The objective is to be able to compute the relevance of any retrieved source code element to the developer, being it in the context model or not. The relevance is measured as a distance between a source code element and the context model of the developer. This distance is computed using the structural and lexical relations that exist between the source code elements, which are represented using an ontology.

Another important difference is that their approach is grounded on the concept of tasks, which requires the developer to explicitly specify when a task starts and ends, and makes it difficult to deal with situations when the developer works on more than one task at the same time. Our approach aims to create a context model that represents the current focus of attention of the developer, independently of the task, or tasks, s/he is working on that moment, thus being more generic. The changes in the focus of attention of the developer are automatically detected by the system, allowing the context model to adapt to the behaviour of the developers without requiring their direct intervention.

In the same line of task management and recovery, Parnin and Gorg (Parnin and Gorg, 2006) propose an approach for capturing the context relevant for a task from a programmer's interactions with an IDE, which is then used to aid the programmer recovering the mental state associated with a task and to facilitate the exploration of source code using recommendation systems. In this scenario, they define context as *"a subset of elements of a program, such as a method, a class or a file, in which the programmer is interested at a given point in time"*. The context information is gathered through the analysis of the interaction history of the user, which is defined as *"a record of a user's interactions with an application for the purpose of providing insight into that history as well as facilitating future interactions"*. They analyze the interactions of a programmer to create a representation of the context of the programmer. The categories of interactions they are interested in are those that apply to an IDE, such as navigation, click, edit and query. The interaction history stores the time and the names of the methods with which the programmer interacted during a work day. Because the contexts derived from large sessions contain too many methods, these are filtered using measures of interest based on the amount of time a programmer spent with a method and the transition patterns between methods.

From the perspective of a context-recommendation system, the objective is to provide a list of items related to an active entity, based on the assumption that accessing an entity from this list is easier than accessing it through other ways. Their idea was to display a list of the active methods in the IDE, and the problem of filtering the most relevant methods to be displayed was resumed to a replacement problem, which they have studied using various replacement algorithms. As a further enhancement, prefetching algorithms were also used, in order to recommend methods that were not previously encountered. They conducted an exploratory case study, obtaining interaction history from professional programmers through a plugin for Microsoft Visual Studio[3] called InteractionHistoryDB, concluding that their approach demonstrates improved exploration of source code over traditional recommendation systems (Parnin and Gorg, 2006).

While their approach focus only on methods, our context model takes into account other elements, such as classes and interfaces. They recommend a list of methods whose relevance is computed based only on the interaction patterns of the developer, ignoring any structural or lexical relations that may exist between them. We use the most relevant and most recently accessed elements in the context model to discover structurally related elements that may be of interest for the developer. Furthermore, we use also the structural and lexical relations between the retrieved elements and the elements in the context model

[3]`http://www.microsoft.com/visualstudio/` (August 2012)

to rank the recommendations according to their relevance to the developer.

Piorkowski et al. (Piorkowski et al., 2011) have studied previous approaches used to help developers navigating the source code structure. These approaches use various models for predicting the locations to where developers need to navigate at a certain moment, including recency, working sets, frequency, bug report similarity, within-file distance, forward call depth, undirected call depth and source topology. They have concluded that recency was the most accurate model for predicting click-based navigations, while within-file distance obtained the best results for view-based navigation. They have also studied a multi-factor model based on Information Foraging Theory (IFT) (Pirolli and Card, 1999), which uses both the source code topology and the words contained in methods. A combination of this model with the recency model provided consistent better results in both click and view-based navigations.

More recently, Piorkowski et al. (Piorkowski et al., 2012) have implemented a new algorithm based on their previous findings (Piorkowski et al., 2011) on using IFT to help developers navigating the source code. They take into account a list of the last methods visited by the developer, which size can be configured based on a low momentum schema (1) and a high momentum schema (10). The algorithm uses this list of previously visited methods to produce a list of 10 recommendations, including methods previously visited and not previously visited by the developer. The recommendations are retrieved and ranked according to a spread-activation algorithm based on a graph representing the source code structure and the words shared between methods. The recency information was also taken into account, by applying a greater activation to more recently accessed methods. This algorithm was compared with a purely lexical-based algorithm that uses the Term Frequency/Inverse Document Frequency (TF-IDF) (Salton and Buckley, 1988) measure to compute the similarity between methods. They implemented a plugin for Eclipse that could provide recommendations using the two algorithms, and conducted a study with a group of professional developers completing a debug task. They analyzed the hit rate and usefulness of the system and discovered that the low momentum schema, using only the last visited method, provides better recommendations than the high momentum schema. They have also found evidence that using the source code structure and the words shared between methods is better that using only the words. Finally, they concluded that the system was able to recommend relevant places for exploration, but that these recommendations were more useful later in the task.

Their approach is comparable with that proposed by Parnin and Gorg (Parnin and Gorg, 2006), as they also focus the methods recently visited by the developer. They extend the approach of Parnin and Gorg, by taking into account the structural relations and lexical similarity between methods to identify a set of relevant methods. Again, our recommendation approach includes classes and interfaces, besides methods, as recommendable items. We measure the lexical proximity between these elements through the co-occurrence of terms in their identifiers, instead of using TF-IDF to find their similarity. Finally, we use all the elements in the current context of the developer to rank the recommendations according to their structural and lexical proximity to the current context.

Singer et al. propose NavTracks (Singer et al., 2005), a plugin for Eclipse that provide recommendations of relevant source code files based on the navigation history of a software developer. The patterns created by the developer when navigating in the source code are used to identify relationships between related files. The association rules are created between files that are accessed in short navigational cycles and are stored in an association repository. When the developer opens or navigates to a source code file, the system collects a list of related files using the association rules stored in the repository. The recommendations retrieved using more recent relationships are ranked higher than the others. The algorithm gives more importance to recent relationships over the more frequent ones,

because a frequency heuristic would not be suitable to deal with task switches.

Deline et al. (DeLine et al., 2005) propose an approach for recommending source code elements based on wear-based filtering, which uses the interaction information of software developers to guide the attention of other software developers exploring the same source code. The current position of the developer in the source code is used as an implicit query for retrieving the source code elements that have been most frequently accessed next. They propose that these frequently accessed next elements could be recommended to the developer in a list, ranked according to how frequently they were accessed.

The RASCAL recommender agent was developed by Mccarey et al. (McCarey et al., 2005) to provide recommendations of relevant methods to a software developer. The system builds an user/item preference database, by mining usage histories of software components in a repository. This database is then used to compute similarities between users and infer relationships between the software components, such as their order of use. The recommender agent monitors the activities of the developer to update the user preferences and recommend methods that are likely to be employed by the developer in a specific context. The recommendations are collected and ordered using both collaborative and content-based filtering techniques.

The last three works we have described are based on the extraction of navigational patterns between the source code elements. These patterns are used to infer association rules that allow to predict a set of potentially relevant elements, based on the current or previous visited elements. We use the interactions of the developer with the source code elements to build a context model representing the source code elements that are more relevant to the developer in a specific moment. But, instead of using navigation patterns, the recommendation of potentially relevant source code elements is based on the structural and lexical relations between these elements and the elements in the context model.

6.2 Software Exploration

A software system is usually modified and extended several times during its lifetime, so that existing features can be improved, new features added and existing bugs resolved. The process of applying such changes to a software system is typically known as software maintenance and software evolution. When performing these change tasks, developers need to explore and investigate the source code to understand which source code artifacts are involved in the implementation of a specific feature. The task of identifying the source code artifacts associated with a feature is known as feature location, or concept location, and is one of the most common activities performed by developers (Dit et al., 2011b). Several approaches have been proposed to help developers locating features in the source code, providing the necessary tools to guide the developers in their exploration. The different techniques applied to feature location are typically categorized as dynamic, static and textual (Dit et al., 2011b). The dynamic approaches help locate features that can be observed during runtime and are typically associated with the analysis of information produced in execution traces. The static approaches explore the structural dependencies of the source code, which are often used by developers when exploring the source code by their own. The textual approaches rely on the words that can be found in the source code, for instance in identifiers and comments, to help locate features. Finally, some approaches apply more that one technique at the same type, trying to overcome the individual limitations of each technique alone. In this section, we will focus on works that apply textual, static or both techniques to feature location, which can be compared with our own approach.

6.2.1 Textual Approaches

The textual approaches to feature location usually rely on one of three techniques to establish a relation between a feature and its location in the source code: pattern matching, Information Retrieval (IR) and Natural Language Processing (NLP) (Dit et al., 2011b). The pattern matching approach provides a way to match specific patterns in the lines of code that comprise a software system. A good example of such an approach is grep[4], an Unix utility for searching plain text with regular expressions. But the pattern matching approaches have several shortcomings, requiring a direct correspondence between the pattern and the text in the source code. In this section, we analyze some approaches that make use of IR, NLP techniques and ontologies. These approaches can be comparable to our own approach, as we also apply an IR technique, to retrieve the most relevant source code components, some concepts of NLP, to identify lexical relations between different source code elements, and ontologies, to represent the source code structure. But our approach is more generic, in the sense that we are addressing search and recommendation of source code in the IDE, independently of the type task being performed by the developer. Also, we are focused on exploiting the context of the developer to enhance the results provided by the IR technique alone. Most of the existing textual approaches ignore, or make little use, of contextual information to help locate features in the source code.

Poshyvanyk et al. (Poshyvanyk et al., 2006b) proposed the integration of the Google Desktop Search (GDS) with Eclipse. They have developed a plugin for Eclipse that allows the developer to perform a search using GDS, the same way they would do with the file search provided in Eclipse, taking advantage of advanced features, such as searching with multiple terms, exact phrases, boolean operators and restrictions on result types. A pilot case study shows that GDS is faster than the Eclipse file search and also easier to use.

Marcus et al. (Marcus et al., 2004) propose an approach to concept location using an IR technique named Latent Semantic Indexing (LSI) (Deerwester et al., 1990). The source code is converted to a corpus, where each function and declaration block is represented as a document. Each document is indexed by a set of terms extracted from the comments and identifiers in the source code represented by that document. The user can formulate queries in natural language, or using terms contained in the source code, and the system retrieves a list of documents ranked according to their similarity to the search query. Their approach was later implemented as a plugin for Eclipse (Poshyvanyk et al., 2006a), allowing the developer to perform fragment-based searches and providing suggestions for formulating the search query. This approach has been augmented by Poshyvanik and Marcus (Poshyvanyk and Marcus, 2007), by applying Formal Concept Analysis (FCA) (Ganter and Wille, 1999) to automatically organize the search results. The FCA technique is used to create concept lattices, using attributes that represent the terms automatically extracted from comments and identifiers in the top ranked search results. These concept lattices are presented to the user, with links to the documents in the source code, and can be used to browse the search results and refine the search query.

Lukins et al. (Lukins et al., 2008) have applied the Latent Dirichlet Allocation (LDA) (Blei et al., 2003) to find the source code entities related to a bug. The source code is automatically processed, using a plugin for NetBeans[5], in order to create a document collection. The terms extracted from the comments, identifiers and strings found in methods are stemmed, cleaned from stop words and associated to a document representing a method. The document collection for all the methods is provided to the LDA tool, which creates a static LDA model of the source code. This model is then queried when a bug is discovered, for instance using the terms extracted from the bug title and description,

[4] `http://www.gnu.org/software/grep/` (August 2012)
[5] `http://netbeans.org/` (August 2012)

which must be processed in the same way as the source code. The system returns a list of methods that may need to be modified in order to solve the bug.

Gay et al. (Gay et al., 2009) propose the use of Relevance Feedback (RF) (Rocchio, 1971), a technique that uses the user input to improve IR algorithms, when locating concepts, or features, in the source code. They combine an IR based approach, similar to the one proposed in (Marcus et al., 2004), with an explicit RF mechanism. During the examination of the search results, the user is asked to evaluate a search result as relevant, irrelevant or neutral. The search query is then reformulated, by removing the terms associated to the set of relevant results, and removing the terms associated to the set of irrelevant results. This way, the query is iteratively refined, in order to guide the user to find the most relevant results.

Shepherd et al. (Shepherd et al., 2007) propose the use of natural language analysis of the source code to help locate and understand the implementation of concepts. Their approach, named Find-Concept, combines structural program analysis with NLP applied to the source code. The source code is modeled using an Aspect-Oriented Identifier Graph Model (AOIG), which represents actions and their direct objects. The retrieval process starts with a query formulated by the user, that must be decomposed in a `Verb-Query` and a `Direct-Object-Query`. Then, the system automatically recommends other `Verb-Query` or `Direct-Object-Query` words, based on the NLP analysis of the source code, that the user may iteratively include in the query. Finally, the search results are presented to the user in the form of a result graph, where nodes represent the retrieved methods and edges represent their structural relationships.

Hill et al. (Hill et al., 2009) propose an approach to support the developer in composing the search queries and evaluating the relevance of the search results. Their approach was inspired by some insights gathered from the work of Shepherd et al. (Shepherd et al., 2007) described before. They concluded that the previous approach was not able to search for features represented as noun phrases, without verbs. This way, they propose to use phrases, or word sequences, extracted from the source code, to capture the context of the words used in the user query. These phrases are automatically extracted from method and field signatures. When the developer performs a search, the search results are grouped according to the phrases to which their are associated, and these phrases are organized in an hierarchy, from most general to more specific phrases.

Abebe and Tonella (Abebe and Tonella, 2010) have applied NLP to extract domain concepts from program element identifiers and organize them in an ontology. The list of terms extracted from a class, attribute or method is parsed and analyzed so that a sentence is generated. The nouns contained in the generated sentences are used to create concepts in the ontology. These concepts are then connected with ontological relations derived from the linguistic dependencies expressed in each sentence. This ontology is then used to reformulate queries for concept location. Their approach was later improved in order to prune the produced ontology from irrelevant implementation details (Abebe and Tonella, 2011). They have studied two IR approaches, one based on term frequencies and other based on topic modeling, to filter irrelevant concepts from the ontology. The two approaches achieved a poor performance, but they have concluded that these automated techniques could be largely improved with a semi-automated approach, requiring a developer to select the most relevant keywords from an automatically generated list of keywords.

In their work, de Alwis and Murphy (de Alwis and Murphy, 2008) try to overcome the need to use several tools to answer conceptual queries posed by a developer. They have developed Ferret, a tool that integrates information from different sources, which are defined as spheres. Answering a conceptual query involves resolving relations between different spheres or composite spheres. Their approach is able to answer 36 different

conceptual questions over four different spheres.

Wursch et al. (Wursch et al., 2010) propose an approach to answer natural language queries about a software system, similarly to Ferret (de Alwis and Murphy, 2008). Their approach was built on top of Evolizer[6], a platform for software evolution analysis that integrates information from different software repositories. An ontology layer was added on top of the Evolizer data layer, comprising an OWL (Bechhofer et al., 2004) ontology that provides a formal description of the semantics associated to the source code. This ontology is used to answer natural language queries about the source code through Ginseng (Guided Input Natural Language Search Engine) (Bernstein et al., 2006), a tool that provides an interface for querying an OWL/RDF knowledge base using quasi-natural language queries.

6.2.2 Static Approaches

The feature location using static approaches follows the usual behaviour of a developer when exploring the source code, using the structural relations that exist between the source code artifacts to guide the investigation process. These approaches usually require the developer to provide a starting point, usually a set of classes and/or methods, and then use the structural relations represented in a model of the source code, such as a call graph, to identify other potentially relevant source code elements. We also use the structural relations of the source code to identify source code elements that may be interesting for the developer in a specific context. But, one big advantage of our approach is that it does not require the developer to provide the initial set of relevant elements, as they are automatically identified by capturing the context of the developer. The structural relations are also used to evaluate the relevance of each retrieved element, in relation to the current context of the developer.

Janzen and Volder (Janzen and De Volder, 2003) developed JQuery, an Eclipse plugin that combines hierarchical browsing with a logical query language to help developers navigate the source code structure. The process starts with a logical query, containing a predicate and a set of variables, that can be created by the developer or picked from a list of predefined queries. The results are displayed in a hierarchy, using the relationships expressed in the query. The hierarchy can then be extended, using other types of relationships that can be selected in context-menu specific to each node.

Saul et al. (Saul et al., 2007) have developed FRAN, a random walking algorithm that uses the structure of object-oriented programs to recommend related functions. Starting from a query function, their algorithm first identifies a set of functions that are in the same layer of the query function, including sibling functions (those that are called by the same functions that call the query function) and spouse functions (those that call the same functions that are called by the query function). The set of functions retrieved in the first step are then ranked according to the concept of authorities and hubs, provided by the HITS[7] (Hypertext Induced Topic Selection) algorithm. A call graph of the functions retrieved is used to find hubs (functions that aggregate functionality) and authorities (functions that implement functionality). They have also developed a second algorithm, named FRIAR (Frequent Itemset Automated Recommender), which uses sets of functions that are frequently called together to mine association rules that predict the most relevant functions associated to a query function.

Robillard (Robillard, 2008) propose an approach to help developers exploring the source code, using the structural dependencies of the source code to identify and rank source code elements that worth investigate. They focus on the problems faced by developers when investigating the source code associated to a task they have to accomplish.

[6] `http://www.evolizer.org` (August 2012)
[7] `http://en.wikipedia.org/wiki/HITS_algorithm` (August 2012)

The elements of interest are explicitly provided by the developer, using concern models created with ConcernMapper (Robillard and Weigand-Warr, 2005), a plugin for Eclipse that allows the developer to specify concerns and associated program elements, such as methods and fields. The structural relations of the elements of interest, more specifically the method call and field access relations, are used to find a set of elements that can be suggested to the developer. The suggested elements are ranked according to a set of heuristics based on the concepts of specificity and reinforcement. The specificity assures that elements having fewer structural relations are more important, while reinforcement increases the importance of elements that are contained in clusters that include other elements of interest. The suggestions are provided to the developer using Suade (Warr and Robillard, 2007), a plugin for Eclipse that integrates with ConcernMapper, in the form of a list of methods ranked according to the degree of membership assigned by the ranking algorithm.

6.2.3 Textual/Static Approaches

Some approaches combine textual techniques with static techniques to improve the feature location process. To the best of our knowledge, none of these approaches makes use of the current context of the developer, relying only in IR techniques to provide an initial set of elements, which are then used to retrieve related elements and rank them according to their structural proximity.

An approach to software exploration using both structural and lexical information is proposed by Hill et al. (Hill et al., 2007). Their tool, named Dora, automatically identifies the relevant neighborhood for a set of seed methods. The seed methods are selected through a search process, initiated with a search query provided by the developer. The candidate relevant methods are identified using the structural call relations with the seed methods. These candidate methods are ranked using a method relevance score, that represents their relevance to the query. The method relevance score is computed using the term frequency, the location of the term in the method and the content of the method. The candidate methods that obtain a score higher than a first threshold are considered relevant, while methods with score between the first threshold and a second threshold are further explored.

Shao and Smith (Shao and Smith, 2009) propose an approach to feature location that combines an IR technique with structural information. Their methodology uses LSI to retrieve a list of methods, and their scores, based on a query. A call graph model is then used to identify all the methods that are structurally related with the retrieved methods. The methods that have a structural relation with other retrieved methods are considered more relevant and have an additional score. The final score of each method is computed as a combination of the scores obtained by the LSI and call graph components.

6.3 Retrieval in Software Reuse

The software reuse (Krueger, 1992) research area has provided several approaches to help software developers in reusing existing source code faster and easier, so that reuse could become a common practice in the programming process. One of the most important steps of the reuse process is to find relevant software components that are suitable for reuse in a specific context. Several techniques have been proposed to create repositories of reusable components and provide the necessary mechanisms to effectively retrieve those components. Some approaches support the retrieval process on the textual references found in the source code, while others explore structural information to evaluate and rank software components according to their applicability in a specific context. Although our

approach is focused on source code exploration (helping developers finding relevant source code elements in the code they are developing or maintaining), some of the approaches that are used in software reuse can be related with our work. For instance, some of them use IR approaches, others apply some heuristics to evaluate the structural proximity between source code elements, and some even use some sort of contextual information to automate and improve the retrieval process. Here we present a set of works that focus on improving the retrieval of software components for reuse in the IDE, grouped according to their retrieval approach, which can be lexical, structural or both.

6.3.1 Lexical Retrieval

The lexical approaches to the retrieval of reusable software components make use of the textual references found in the source code to locate potentially reusable source code. These approaches compare with our own in the sense that they use this textual references to retrieve source code, although this is done with a very different purpose. Despite the fact that some of these approaches use the current context of the developer to improve the retrieval, or even to trigger the suggestion, of reusable components, the context is usually limited to the current active method or class, while our approach makes use of a more complex context model that represents the current focus of attention of the developer.

The CodeFinder tool, proposed by Henninger (Henninger, 1996), uses an approach based on an associative spread activation retrieval algorithm and query reformulation, to improve the retrieval of software components and reduce the effects of indexing problems. The software components are automatically associated to individual terms in an associative network, which is then used by the spread activation algorithm to induce semantic relationships between components that share the same terms, and terms that co-occur in the same component. The query reformulation mechanism provides suggestions of relevant components and terms that can be included in the query. The system also provides a way to improve the repository structure, by allowing the developer to assign new terms to a component and using relevance feedback to learn the weight of the relations that link terms to components.

Ye and Fischer (Ye and Fischer, 2002) propose CodeBroker, a tool that pro-actively suggests reusable components to Java developers using Emacs[8]. The system creates an index of the Java documentation (JavaDoc) associated to Java source code. An interface agent monitors the JavaDoc comments and signature definitions created by the developer, from which it extracts queries to retrieve matching components. The components retrieved are ranked according to their relevance, and are filtered from components contained in a discourse model and an user model. The discourse model includes components that have been explicitly marked by the developer as not being interesting for the current session. The user model contain components that are already known by the developer.

Chatterjee et al. (Chatterjee et al., 2009) propose the SNIFF search algorithm, which combines Application Programming Interface (API) documentation with Java source code to retrieve chunks of source code that match a free-form query. The source code lines are automatically annotated with the documentation and terms, extracted from the APIs that are used in each line, and then indexed in a database for retrieval. When the developer performs a search, the system retrieves source code chunks annotated with words that match the words contained in the query. The source code chunks retrieved are grouped in clusters, based on their similarity. The clusters presented to the developer are ranked according to their frequency in the code base.

Heinemann et al. (Heinemann et al., 2012) propose an approach to recommend methods that may be relevant for the current work of a developer. They try to overcome the

[8] `http://www.gnu.org/software/emacs/` (August 2012)

problem of lack of information faced by structural approaches, and provide recommendations based on the terms extracted from identifiers preceding method calls. They use the terms contained in the identifiers that precede method calls to create an associative index between a set of terms and the method call. This index is queried with the terms contained in the identifiers that precede the current position of the cursor in a source code file, to retrieve API methods that have the most similar term sets.

6.3.2 Structural Retrieval

The retrieval of reusable software components based on the structural relations found in the source code has been explored in various ways. The structural nature of source code has been used to compute the similarity between different source code artifacts, to find the source code needed to go from an object to another or to instantiate a specific type of object, and to find source code that fit on specific test or design specifications. Our approach focuses on finding the source code elements that are more relevant in the current context of the developer, using the source code structure to compute the proximity between the retrieved elements and those in the current context model of the developer. The use of context on these approaches is once again typically limited to the current position in the source code or to a set of seed elements provided by the developer.

Strathcona, an Eclipse plugin proposed by Holmes and Murphy (Holmes and Murphy, 2005), is used to help developers locating source code examples that are relevant for their current task. When the developer requests for examples related with a class, method or field declaration, the system generates a structural context of these elements to find source code examples that match that description. The structural context depends on the type of element in the query, and may include the containing class, parent classes or interfaces, the types of the fields of these classes and calls of the queried method. The match between the structural context and the source code in the repository is computed using six heuristics based on inheritance, method calls and field types.

Mandelin et al. (Mandelin et al., 2005) developed an Eclipse plugin, called PROSPECTOR, to assist developers that need to obtain a specific type of object, but do not know how to write the code needed to obtain that object. Their approach accepts queries in the form of input and output types. These queries are answered with a ranked list of jungloids (a list of objects and method calls needed to go from an object type to another), that are mined through the analysis of API signatures. The tool is integrated with the content assistant feature of Eclipse, automatically providing suggestions of jungloids that can be used to obtain an object type found in the current context, which comprises the source code surrounding the current position of the cursor.

Similarly to PROSPECTOR, the PARSEWeb tool, proposed by Thummalapenta and Xie (Thummalapenta and Xie, 2007), suggests examples of method call sequences, helping developers going from a specific object type to a desired one. Instead of using only the API signatures, PARSEWeb relies on entire code samples to answer a query.

With the objective of assisting developers with examples in the form of source code snippets, Sahavechaphan and Claypool (Sahavechaphan and Claypool, 2006) proposed an Eclipse plugin called XSnippet. The tool provides help in object instantiation tasks, supporting simple constructor invocations, static method invocations and more complex sequence of method invocations. The snippets can be retrieved using generalized, type-based and parent-based instantiation queries. A generalized instantiation query returns all the snippets that instantiate a given type. The other two types of queries are context dependent. The first takes into account all the types that are related with the current method, while the second takes into account the parent types of the type containing the current method. The retrieved snippets are ranked using three different heuristics: length,

frequency and context. The length heuristic ranks snippets based on the lines of code, snippets with fewer lines of code are ranked higher. The frequency heuristic assures that snippets occurring more frequently are ranked higher. The context heuristic ranks snippets according to the match between the context of the current method and the context of the snippet.

Hummel et al. (Hummel et al., 2008) developed an Eclipse plugin for retrieving relevant software components, called Code Conjurer. The tool aims to proactively provide useful software reuse recommendations, using a test-driven and a design-based search approaches. The test-driven approach provides recommendations upon the definition of a test case, and assures they are useful for the developer by testing them with the defined test case. The design-based approach retrieves implementation recommendations for the components being designed by the developer. Finally, it also provides an automated dependency resolution feature, to automatically fetch components that are required by a component being reused.

Zhong et al. (Zhong et al., 2009) developed an Eclipse plugin called MAPO, which automatically mines and recommends API usage patterns to assist developers with useful source code snippets. The recommendation is performed when the developer requests API patterns related with a method. The recommended patterns are ranked according to the current context of the developer, by computing a similarity value between the names of current active method and class, and the names in the recommended code snippet.

Keivanloo et al. (Keivanloo et al., 2010) proposed a Semantic Web (Berners-Lee et al., 2001) approach to code search with SE-CodeSearch, an Internet-scale source code search infrastructure. Their approach uses an OWL (Bechhofer et al., 2004) ontology to model the source code structure and applies Semantic Web reasoning to infer missing knowledge and answer complex queries. The ontology was designed to deal with high-level concepts of object-oriented source code, including the type hierarchies, package relationships and their dependencies, method calls and return statements, and unqualified name resolution. The infrastructure provided by this ontology, and associated reasoning mechanisms, is able to answer pure structural queries, metadata queries, transitive queries, method call-based queries, absent information queries and mixed queries.

6.3.3 Lexical/Structural Retrieval

The lexical and structural approaches have also been combined, in order to overcome the limitations of both approaches alone and improve the retrieval process. Although with a different purpose, we also combine the lexical a structural perspectives of the source code to compute the proximity between the retrieved source code elements and the current context model of the developer.

Exemplar, created by Grechanik et al. (Grechanik et al., 2010), combines IR and program analysis techniques to retrieve relevant applications for tasks or requirements expressed through high level queries. Their approach makes use of the help documentation of APIs, and the dataflow links between them, to improve the retrieval and ranking of relevant applications. The keywords from a query are matched in the description of the applications and in the help documentation of API calls invoked by the applications. The retrieved applications are then ranked according to a set of weights that represent three components: the matching between the keywords in the query and the description of the application, the matching between the retrieved API calls and those that are invoked by the application, and the dataflow connections between the API calls in the application.

Portfolio, proposed by McMillan et al. (McMillan et al., 2011), is a search engine that combines different models to improve the retrieval and visualization of functions and their usages. The relevant functions are retrieved using a Vector Space Model (VSM)

approach, by matching the query terms with the terms in the source code. A navigation model, based on the PageRank (Brin and Page, 1998) algorithm, is used to rank functions according to the query terms and functional dependencies that are shared between them. An association model, based on a Spread Activation Network (SAN) (Collins and Loftus, 1975), uses the function call graph to propagate the score of relevant functions to related functions, where the number of shared terms reflects the strength of the relation between them. The final ranking of each function is computed as a weighted sum of the scores obtained through the navigation model and the association model. The search results are presented to developers in a list and in a graph where the usage relations between them can be visualized. Their experiments show evidence that this approach performs better that Google Code Search and Koders[9], and that the visualization mechanism allow the developers to understand how the retrieved functions are used.

6.4 Software Project History

The history associated with a software development project is typically stored in various systems, such as version control systems and issue tracking systems, and different types of resources, such as messages or documents, etc. These systems and resources represent a valuable source of information about a software system, and have been used to uncover implicit knowledge and provide guidance to software developers. Here we present a set of works that have exploited the historical information associated to a software development project to retrieve relevant source code elements for a software developer. They are comparable to our approach, in the sense that they provide mechanisms to retrieve relevant files or source code elements based on a query or a specific context, which is typically reduced to current active file or source code element. But, while they use the contextual information provided by the historical archive of a software system, we rely on a context model based on the current interactions of the developer with the source code, along with their structural and lexical relations, to identify and retrieve the most relevant source code elements for the developer in that specific moment.

Ying et al. (Ying et al., 2004) have developed an approach that uses the information provided by a software configuration management system, such as Concurrent Versioning System (CVS)[10], to identify relevant source code to a developer performing a modification task. Their approach identifies change patterns by mining the information provided by such systems with Data Mining (DM) (Witten and Frank, 2005) techniques. These change patterns are used to recommend relevant source code files contained in the same change sets that include an initial source code file.

Following the same concept, ROSE (Zimmermann et al., 2005), developed by Zimmermann et al., provide recommendations of more fine-grained entities, such as source code elements, that were changed together. Their approach uses DM techniques to extract association rules from the changes applied to the source code in a CVS archive. The association rules extracted are then used to predict further changes, reveal hidden couplings and avoid errors produced by incomplete changes.

Hipikat was developed by Cubranic et al. (Cubranic et al., 2005) to recommend relevant artifacts from a project memory. The project memory is made up of the various kinds artifacts created during a software development project and their relationships, including source code, documentation, email messages, forum posts, bug reports and test plans. The artifacts and relationships contained in the project memory are automatically extracted from the project archives used in a software development project. The recommendations

[9] `http://www.koders.com/` (August 2012)
[10] `http://en.wikipedia.org/wiki/Concurrent_Versions_System` (August 2012)

of relevant artifacts are provided upon a query from the developer. These recommendations are retrieved by following the relationship links between the artifacts of interest and other artifacts.

More recently, Ratanotayanon et al. (Ratanotayanon et al., 2010) proposed the concept of transitive changesets to uncover transitive relationships that can be used to help developers identifying features in the source code. A changeset contains information about atomic changes performed in a revision control system and some metadata. The transitive changesets are created by expanding the information associated to changesets with transitive relations, including, for instance, which lines of code, and respective source code elements, were changed in a changeset. They have developed a plugin for Eclipse, called Kayley, which allows the developer to query a repository of transitive changesets extracted from a commit history and obtain a list of the program elements changed in the retrieved changesets.

Chapter 7

Conclusions

"Life is the art of drawing sufficient conclusions from insufficient premises."

Samuel Butler

The research work described in this thesis represents the steps taken towards the fulfillment of the objectives that were set in the beginning of our journey. We proposed to use the contextual information of the developer to improve the retrieval of relevant source code artifacts during software development. This high level research goal was decomposed in a set of more fine grained objectives, including the definition of a context model of the developer, the use of this context model to improve the ranking of source code elements searched by the developer, and to pro-actively provide recommendations of relevant source code elements to the developer.

The approach we have proposed for achieving our objectives began with the definition of a *knowledge base*, where the source code structure stored in the workspace of the developer is represented. This knowledge base comprises a structural ontology, which is used to make explicit the different types of source code elements and their relations, and a lexical ontology, which represents the terms used to reference the source code elements.

The contextual information of the developer is modeled in the form of a *context model*. As in the knowledge base, this context model combines a structural and a lexical dimensions, which represent the source code elements, their structural relations and terms, that are more relevant for the developer in a specific moment. A context transition detection mechanism allows the context model to automatically adapt to the changes in the focus of attention of the developer.

The context model defined is used to support a *context-based search* process, in which the results are retrieved using an Information Retrieval (IR) model and ranked according to a retrieval, a structural and a lexical components. The retrieval component represents the ranking provided by the IR model, while the structural and lexical components represent the proximity of the search result to the context model of the developer. The contribution of these components to the ranking of search results is defined by a set of weights, which are learned over time, through the analysis of the search results selected by the developer.

This context model was also used to support the *context-based recommendation* of relevant source code elements to the developer. The recommendations are retrieved using the source code elements in the context model with higher interest and accessed more recently. The retrieved recommendations are then ranked according to an interest and a time components, representing the ranking obtained through the retrieval process, as well as a structural and lexical components, which represent the proximity of the recommendation in relation to the context model. As in the context-based search process, the weights

of these components are learned over time, through the analysis of the recommendations selected by the developer.

We have implemented a *prototype* that implements and integrates our approach in the Eclipse IDE. This prototype was tested with a group of developers in order to validate our approach. The statistical information collected shows that the source code elements manipulated by the developer are highly related, being structurally or lexically related with other elements already in the context model in more than 80% of the times. This supports our claim that the relations that exist between source code artifacts can be used to measure the proximity between these artifacts and to compute their relevance in the current context of the developer. Also, we have verified that the context components have a clear contribution to improve the ranking of search results. The search results selected by the developers using our approach, were better ranked in about 60% of the times, and worst ranked in only 11% of the times. With respect to recommendations, although the results are not so evident, we have shown that our context model could be used to retrieve relevant source code elements for the developer, being able to predict the needed source code element among the top 30 recommendations in about 53% of the times.

The main contributions of this research, including a list of scientific publications that were produced, are described in the following section. Then, the chapter concludes with a set of improvements, suggestions and questions that remain open for future work.

7.1 Contributions

The research developed during this thesis resulted in several contributions, including improvements over previous work, new approaches for problems yet to be solved and a prototype that is publicly available. Here we will describe each one of these contributions, concluding with a list of scientific publications that are within the scope of this thesis.

The *context model of the developer* is in the basis of all the research we have conducted (see section 3.2). Although the context model we have developed was inspired by previous work (Kersten and Murphy, 2006), it introduces innovations that were not considered before. We have extended the previous model with a lexical perspective (see section 3.2.2), which allowed us to explore the lexical relations between the source code elements, the same way that the structural relations were explored before. This lexical perspective has revealed to be relevant in the ranking of search results, and especially recommendations. We believe that the potential of the lexical relations in the source code was not fully explored, because much more is yet to be improved, investigated and evaluated, as described further in future work.

Additionally, we have also developed a *mechanism to automatically detect context transitions* (see section 3.2.3). The aim of this mechanism is to detect changes in the focus of attention of the developer and reflect those changes in the context model. We wanted to avoid the explicit association between the context model of the developer and tasks, because they are difficult to manage and require the explicit intervention of the developer. The mechanism we have developed allows the context model to be adapted faster and more close to the behaviour of the developer, without requiring any kind of direct intervention. This mechanism was especially important for the context-based recommendation process, because it is completely dependent on the context model for retrieving and ranking recommendations.

The aforementioned context model was used to support an *approach to context-based search of source code in the IDE* (see section 3.3). The search results are retrieved using an IR model, and are ranked according to both this retrieval model and the context model of the developer. This way, the ranking of a search result is influenced by its proximity to

the context model of the developer, tanking into account both the structural and lexical relations that exist in the source code. To the best of our knowledge, this is the first approach to combine IR and a context model of the developer to retrieve and rank source code elements in an IDE. As the evaluation of our approach has demonstrated, the use of the context model provides a clear improvement in the ranking of search results.

Moreover, we have also developed an *approach to context-based recommendation of source code in the IDE* (see section 3.4). The elements in the context model with higher interest and that have been accessed more recently are used to retrieve recommendations of relevant source code elements to the developer. These recommendations are then ranked according to the elements that got them retrieved, as well as to their structural and lexical proximity to the entire context model. Although other works have tackled the problem of predicting the needs of the developer using similar concepts (Kersten and Murphy, 2006; Parnin and Gorg, 2006; Piorkowski et al., 2012), to the best of our knowledge, our approach is the first that combines the recommendation of different types of elements (classes, interfaces and methods), the use of lexical relations based on the co-occurrence of terms, and the ranking of recommendations taking into account their proximity to the entire context model.

A *learning mechanism* was developed, so that the ranking of search results and recommendations could be adapted to the needs of the developer. This mechanism uses the rankings of the search results and recommendations, selected by developers, to favor the components that have a positive influence in their final ranking. We believe that this mechanism represents a contribution, as we have seen nothing similar applied to the same problems we are addressing.

The context-based search and recommendation approaches developed were implemented and integrated in the Eclipse IDE, using a plugin named *Software Development in Context (SDiC)*. This prototype was used to validate our approach with developers in a real word scenario, so it was developed with particular care with respect to several aspects. We devoted especially attention to performance, stability, and usability, so that the prototype could be used by the developers in their daily work. The prototype was used by a considerable number of developers, whose feedback was used to correct several problems and make important improvements. The result is a prototype that can be easily installed in any Eclipse instance, providing instant access to context-based search and recommendation of source code in the IDE. This prototype is publicly available for download through the web site of the SDiC project (`http://sdic.dei.uc.pt`), representing also an important contribution of our research.

The contributions of this work are described in several scientific publications, presented in both national and international events, including some highly selective ones. Next, we enumerate these publications and provide additional information, when available, regarding the type of publication, the acceptance rate, and the ERA[1] ranking of the event where it was presented.

- Antunes, B. and Gomes, P. (2009). Context-Based Retrieval in Software Development. In *Proc. of the Doctoral Symposium on Artificial Intelligence (SDIA 2009) of the 14th Portuguese Conference on Artificial Intelligence (EPIA 2009)*, pages 1–10, Aveiro, Portugal *(Doctoral Symposium)*

- Antunes, B., Correia, F., and Gomes, P. (2010). Towards a Software Developer Context Model. In *Proc. of the 6th International Workshop on Modeling and Reasoning in Context (MRC 2010) of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 1–12, Lisbon, Portugal *(Workshop)*

[1] `http://core.edu.au/index.php/categories/conference%20rankings/1` (August 2012)

- Antunes, B., Cordeiro, J., Costa, P., and Gomes, P. (2011). Using Contextual Information to Improve Awareness in Software Development. In *Proc. of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 349–352, Miami, USA *(Short Paper — Acceptance Rate: 31% — ERA Ranking: B)*

- Antunes, B., Cordeiro, J., and Gomes, P. (2012d). SDiC: Context-Based Retrieval in Eclipse. In *Proc. of the Informal Demonstrations of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 1467–1468, Piscataway, NJ, USA. IEEE Press *(System Demonstration)*

- Antunes, B., Cordeiro, J., and Gomes, P. (2012c). Context Modeling and Context Transition Detection in Software Development. In *Proc. of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 477–484, Rome, Italy *(Full Paper — Acceptance Rate: 11% (Full Papers) 43.3% (Global) — ERA Ranking: B)*

- Antunes, B., Cordeiro, J., and Gomes, P. (2012b). Context-Based Search in Software Development. In *Proc. of the 7th Conference on Prestigious Applications of Intelligent Systems (PAIS 2012) of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 937–942. IOS Press *(Full Paper — Acceptance Rate: 28.5% — ERA Ranking: A)*

- Antunes, B., Cordeiro, J., and Gomes, P. (2012a). An Approach to Context-Based Recommendation in Software Development. In *Proc. of the 6th ACM Conference on Recommender Systems (RecSys 2012)*, pages 171–178, New York, NY, USA. ACM Press *(Full Paper — Acceptance Rate: 20% — ERA Ranking: B)*

- Antunes, B., Cordeiro, J., and Gomes, P. (2013). An Approach to Context Modeling in Software Development. In *Software Paradigm Trends*, Communications in Computer and Information Science (CCIS). Springer-Verlag, Berlin, Heidelberg *(ICSOFT 2012 Extended Paper — Submitted for Publication)*

7.2 Future Work

We believe that our approach is a relevant contribution on how to use contextual information to improve the retrieval and ranking of relevant source code elements in the workspace of the developer. However, it comprises several components and different processes, whose complexity requires them to be studied in more detail, still leaving plenty of room for improvements. Although we have included a lot of enhancements during the implementation and validation of our approach, we had several ideas and received a number of suggestions that could not be fully explored during the period of this research. This work was built upon the work of others before us, and we know that it was just one more step of a long walk that is yet to come. Here we will discuss some of the ideas, suggestions and questions that we believe should be explored in the future, either by us or any other researches that may be interested in using or expanding our approach.

7.2.1 Knowledge Base

The knowledge base is an essential building block of our approach and is highly dependent on the source code structure created by developers. The quality of the knowledge base is directly dependent on the way developers create and organize their source code (see

section 3.1.2). For instance, if developers do not follow the established naming conventions when defining identifiers, or simply do not use coherent approaches to build quality identifiers, our assumption that we can find lexical relations between source code elements through their identifiers become compromised. This way, the lexical ontology is particularly influenced by the quality and readability of the source code being represented. In fact, several studies have been carried out to evaluate the quality, importance and impact of identifiers in several software development activities (Takang et al., 1996; Caprile and Tonella, 1999; Lawrie et al., 2006; Dit et al., 2011a). This problem could be partially mitigated, for instance, by using enhanced approaches to extract terms from identifiers. Although the CamelCase approach to identifier splitting is the most obvious and widely used approach, more complex and efficient approaches have been proposed in the recent years (Enslen et al., 2009; Lawrie et al., 2010; Guerrouj et al., 2011). The set of terms that index a source code element could also be expanded, for instance, by taking into account the comments associated to that element. Although not all terms contained in these comments may be of interest to index a source code element, the most discriminating terms could be found, for instance, using term frequency based approaches such as Term Frequency/Inverse Document Frequency (TF-IDF) (Salton and Buckley, 1988).

Another problem we may have with terms is that different forms of a same word may be used in different situations, leading to the coexistence of different representations, or terms, of the same word in the lexical ontology. This happens because words have several inflected forms, which are used to express different grammatical categories, such as tense, number or gender. This characteristic of words make it more difficult to identify lexical relations between the source code elements, because we can not create association relations between different terms, although they may represent the same original word. This problem has been addressed in both computational linguistics and information retrieval by using mechanisms that reduce the different forms of a word to a shared primitive form, such as *stemming* (Lovins, 1968). These mechanisms could also be integrated in our approach, so that terms could be stored using their root form, and association rules could be created using these root forms, instead of the original form found in the source code.

The lexical ontology could also be extended by using word synonyms, so that instead of terms, we could have sets of terms that convey the same natural language concept. This could be achieved by using existing lexical ontologies, such as WordNet (Miller, 1995; Fellbaum, 1998), where terms are already grouped in groups of synonyms. But, such approach would require a term found in a source code element to be mapped to its corresponding concept, in what is known as *word sense disambiguation.* Although the mapping between a term and its sense is not a trivial task, especially when we have little linguistic information about the context where the term is used, this approach would allow us to associate source code elements to a network of concepts, instead of simple terms, for instance as proposed by Ratiu et al. (Ratiu and Deissenboeck, 2006). By using concepts instead of terms, we could also extend the lexical ontology with other types of semantic relations, such as *hypernymy* (a concept is a kind of another concept) and *part-of* (a concept is a part of another concept), which can also be found in WordNet.

With respect to the structural ontology, the representation of the source code could benefit from using other sources of information. We could, for instance, process the source code comments, which are known as JavaDoc[2] in the Java programming language, to extract relations between certain source code elements. As an example, we have the `@see` and `@link` tags in JavaDoc, which can be used by the developer to make explicit references from a source code element to other related elements. The relations expressed by these tags may not become explicit through the analysis of the source code structure,

[2]`http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/javadoc.html` (August 2012)

but could be inferred by the analysis of the JavaDoc associated to the respective elements.

The structural ontology could also be complemented with relations extracted directly from the information represented in the context model of the developer. For instance, two source code elements that are part of the same context model are expected to have some relation between them, even if this relation is not explicit in the source code structure. This can not be taken for granted, because the simple fact that the developer accessed these two elements in the same context model does not guarantee that they are related. But, if we also take into account the degree of interest of the two elements, we may use it to measure the likelihood that the two elements were not accessed in the same context model by accident. We may assume that if the two elements have an high interest value, they are both relevant in the current context model and there is some kind of relation between them that can not be ignored. This situation is expected to happen when the two elements have some kind of strucural relation, but when that is not the case, a special relation can be created between them. These type of association rules have been explored before, for instance taking into account the navigation patterns of the developers (Singer et al., 2005; DeLine et al., 2005; McCarey et al., 2005) and the source code artifacts that are changed together in version control systems (Ying et al., 2004; Zimmermann et al., 2005).

7.2.2 Context Model

With respect to the context model of the developer that we have used in or approach, there are a set of processes and parameters of special interest that could be studied in more detail. For instance, the decay applied to the interest of a structural element in the context model follows a linear function of time (see section 3.2.1). We believe that the interest of such elements should be decreased according to the time passed since they were last accessed, thus their interest is decreased at fixed intervals of time. But, should the interest of an element be decayed exponentially, for instance as proposed by Parnin ang Gorg (Parnin and Gorg, 2006) in their momentum function? Additionally, it would be interesting to understand if different decay rates should be applied to different source code elements. For instance, should the interest of a class decay at the same rate as the interest of a method? The answers to these questions are hard to find, leaving room for further investigation.

The weight of the structural relations in the context model represents the weight of a generic type of relation, instead of representing the weight of a specific relation between two structural elements, which was proposed by Kersten and Murphy (Kersten and Murphy, 2006) in their context model (see section 3.2.1). We used the weight of generic relations, because we needed to compute the cost of paths between any arbitrary elements. If we had used a model where only the specific relations between the structural elements in the context model were considered, most of the relations found in these paths would not have an associated weight. But, there are situations when the specific relations between structural elements in the context model will be part of the paths between the structural elements whose distance is being measured. In such cases, the weight of the specific relations could be boosted, so that the weight of such relations could be higher than the weight of the generic relation that connect the two elements.

The context transition detection is another point that requires further research (see section 3.2.3). The parameters that regulate this process were defined based on our own experience and observations, and the results obtained through the developers evaluation of context transitions were not conclusive. The objective of context transitions is to model the change in the focus of attention of the developer, which does not necessarily mean a transition to a new task. This definition of context transition raises some ambiguity,

because we must take into account that a context transition is perceived differently by different developers in different situations. This way, we have also asked ourselves if the parameters that control context transitions should change according to the developer, and even to the situations. Also, our context transition mechanism is entirely based on the structural relations between source code elements. But, it would be interesting to evaluate what is the role of lexical relations in this process. Could lexical relations be used to detect context transitions, or, at least, influence the decision of making a context transition? As we can see, there is still much to be studied concerning the context transition mechanism we have proposed.

7.2.3 Context-Based Search

With regard to the indexing and retrieval processes of the context-based search (see sections 3.1.3 and 3.3.1), the indexing of the source code elements could be improved by using some of the techniques that were already described in section 7.2.1, such as stemming and synonymy. The use of stemming in source code search has been already studied by Wiese et al. (Wiese et al., 2011), showing that stemming can be used to improve the ranking of relevant search results. The use of synonymy would allow us to overcome the problem of a restricted vocabulary. Because the source code elements are indexed using only the terms found in the identifiers of such elements, the search process is limited by the terms used by developers. By including synonyms in the retrieval process, we would expand the vocabulary that could be used, allowing a source code element to be retrieved by the terms in its identifier and also their synonyms. Additionally, we could index the source code elements using terms extracted from other sources of textual information, such as comments and literals. Concerning the IR process, we could explore the use of other IR models that have already been used in the past, such as Latent Semantic Indexing (LSI) (Marcus et al., 2004; Poshyvanyk and Marcus, 2007; Shao and Smith, 2009) and Latent Dirichlet Allocation (LDA) (Lukins et al., 2008).

The process of measuring the proximity between a search result and the context model should also be studied in more detail in the future. The distance between a source code element and the context model is computed using the cost of the shortest paths between that element and all the elements in the context model (see section 3.3.2). This distance is also proportional to the interest of the relations that comprise each path and to the interest o the element in the context model. This approach takes into account all the paths that connect the structural element with the context model. This means that a search result that is related with several elements in the context model may be considered more distant to the context model than another search result that is related with only one element, depending on the interest associated to the structural elements and relations in the context model. But, one can argue, for instance, that a search result that is related with more elements in the context model should be considered more close to the context model than another that is related with only one element. Furthermore, there may be several paths between the same two structural elements, although we only use the shortest path between them in our approach. We may also consider taking into account the several paths that exist between two structural elements when computing the distance between the two. These assumptions are extensible to the lexical ontology, where a similar approach is used. We believe that our strategy is the one that better reflects the proximity between a search result and the entire context model. But, the effectiveness of different approaches must be studied, in order to find which approach obtains better results.

Still concerning the proximity between a search result and the context model, the number of context elements taken into account when computing this proximity was limited to 15, while the paths were limited to a maximum of 3 relations (see section 3.3.2). These

limits were ultimately defined by the need of keeping the computation time within an acceptable time frame. Further investigation is needed, in order to evaluate if different limits have a significant influence in the precision of the search process.

7.2.4 Context-Based Recommendation

Concerning the retrieval process of the context-based recommendation (see section 3.4.1), we have used only the structural context to retrieve the source code elements that are recommended to the developer. We retrieve the top N elements in the structural context, ordered by both their interest and the time their were last accessed, as well as all the elements that are structurally related with these elements. But, it would be interesting to evaluate if the lexical context and the lexical relations could also be used to retrieve relevant recommendations. For instance, we could retrieve the structural elements that are lexically related with the top N elements in the structural context. Also, we could use the top N terms in the lexical context to retrieve other source code elements that were indexed by these terms.

With regard to the number of elements of the context model used to retrieve recommendations, we always retrieve the top N elements, ordered by the interest value and the time elapsed since they were last accessed. We would like to evaluate if using different values of N for the interest and time based methods could improve the precision of the recommendation process.

The user interface used for the context-based recommendation could also be improved, for instance by providing an explanation for each recommendation. These explanations could help developers understand if a given recommendation is interesting or not. We could use the structural elements and relations that contributed for retrieving a recommendation to compose an explanation of that recommendation to the developer. The challenge here is how to include such information in a user interface that is expected to be simple, intuitive and easy to use.

Finally, the same investigations that were suggested for improving the process of computing the proximity between a search result and the context model, apply to the context-based recommendation as well (see section 7.2.3).

7.2.5 Weight Learning

The learning process focuses on learning the best combination of weights that are used in the ranking of search results and recommendations, which will be generically referred as results (see section 3.5). These weights are learned through the analysis of the results selected by the developer. The learning process is processed continually over time, adapting to favor the components that have a positive influence in the ranking of the results that are more relevant for the developer. It would be interesting to evaluate the use of different sets of weights per context model. This approach would be based on the assumption that in different contexts, the components that are more relevant for the developer may change. Thus, why not learn the best weights that should be used in a specific context model, instead of using a set of weights that is independent of the current context of the developer?

7.2.6 Application Domain

Although our context model and context-based retrieval approaches were applied to the software development domain, we believe that the conceptual framework of our work could also be applied to other domains. This would be possible in a scenario where the

user deals with resources that can be related with some sort of relations and that can also be associated with textual references. The structural ontology could be adapted to represent the new kind of resources, along with their structural relations, while the lexical ontology would still represent terms, and their co-occurrence relations, extracted from these resources. The context model of the user would be identical to the one we have used, but the interactions used to create the model could be adapted to reflect the behaviour of the user in this new scenario. In the software development domain, the structural context seems to have a higher relevance than the lexical context, but in a different domain may occur just the opposite. Whatever the case, our weight learning mechanism would guarantee that this particularities would be taken into account.

References

Abebe, S. L. and Tonella, P. (2010). Natural Language Parsing of Program Element Names for Concept Extraction. In *Proc. of the IEEE 18th International Conference on Program Comprehension (ICPC 2010)*, pages 156–159, Washington, DC, USA. IEEE Computer Society.

Abebe, S. L. and Tonella, P. (2011). Towards the Extraction of Domain Concepts from the Identifiers. In *Proc. of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 77–86, Washington, DC, USA. IEEE Computer Society.

Adomavicius, G. and Tuzhilin, A. (2005). Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749.

Adomavicius, G. and Tuzhilin, A. (2011). Context-Aware Recommender Systems. In *Recommender Systems Handbook*, pages 217–253. Springer US.

Anand, S. and Mobasher, B. (2007). Contextual Recommendation. In *From Web to Social Web: Discovering and Deploying User and Content Profiles*, volume 4737 of *Lecture Notes in Computer Science*, pages 142–160. Springer-Verlag, Berlin, Heidelberg.

Antunes, B., Cordeiro, J., Costa, P., and Gomes, P. (2011). Using Contextual Information to Improve Awareness in Software Development. In *Proc. of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 349–352, Miami, USA.

Antunes, B., Cordeiro, J., and Gomes, P. (2012a). An Approach to Context-Based Recommendation in Software Development. In *Proc. of the 6th ACM Conference on Recommender Systems (RecSys 2012)*, pages 171–178, New York, NY, USA. ACM Press.

Antunes, B., Cordeiro, J., and Gomes, P. (2012b). Context-Based Search in Software Development. In *Proc. of the 7th Conference on Prestigious Applications of Intelligent Systems (PAIS 2012) of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 937–942. IOS Press.

Antunes, B., Cordeiro, J., and Gomes, P. (2012c). Context Modeling and Context Transition Detection in Software Development. In *Proc. of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 477–484, Rome, Italy.

Antunes, B., Cordeiro, J., and Gomes, P. (2012d). SDiC: Context-Based Retrieval in Eclipse. In *Proc. of the Informal Demonstrations of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 1467–1468, Piscataway, NJ, USA. IEEE Press.

Antunes, B., Cordeiro, J., and Gomes, P. (2013). An Approach to Context Modeling in Software Development. In *Software Paradigm Trends*, Communications in Computer and Information Science (CCIS). Springer-Verlag, Berlin, Heidelberg.

Antunes, B., Correia, F., and Gomes, P. (2010). Towards a Software Developer Context Model. In *Proc. of the 6th International Workshop on Modeling and Reasoning in Context (MRC 2010) of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 1–12, Lisbon, Portugal.

Antunes, B. and Gomes, P. (2009). Context-Based Retrieval in Software Development. In *Proc. of the Doctoral Symposium on Artificial Intelligence (SDIA 2009) of the 14th Portuguese Conference on Artificial Intelligence (EPIA 2009)*, pages 1–10, Aveiro, Portugal.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Balabanović, M. and Shoham, Y. (1997). Fab: Content-Based, Collaborative Recommendation. *Commun. ACM*, 40(3):66–72.

Basu, C., Hirsh, H., and Cohen, W. (1998). Recommendation as Classification: Using Social and Content-Based Information in Recommendation. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI '98/IAAI '98)*, pages 714–720, Menlo Park, CA, USA. AAAI Press.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). OWL Web Ontology Language Reference. Published: W3C Recommendation.

Belkin, N. J. and Croft, W. B. (1992). Information Filtering and Information Retrieval: Two Sides of the Same Coin? *Communications of the ACM*, 35(12):29–38.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284:34–43.

Bernstein, A., Kaufmann, E., Kaiser, C., and Kiefer, C. (2006). Ginseng a guided input natural language search engine for querying ontologies. In *Jena User Conference*.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022.

Breese, J. S., Heckerman, D., and Kadie, C. (1998). Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proc. of the 14th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 43–52, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Brewster, C., Iria, J., Ciravegna, F., and Wilks, Y. (2005). The Ontology: Chimaera or Pegasus. In *Proc. of the Dagstuhl Seminar on Machine Learning for the Semantic Web*, pages 13–18.

Brin, S. and Page, L. (1998). The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117.

Brown, P. J., Bovey, J. D., and Chen, X. (1997). Context-Aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications*, 4:58–64.

Burke, R. (2002). Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370.

Büttcher, S., Clarke, C. L. A., and Cormack, G. V. (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press.

Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., and Li, H. (2008). Context-Aware Query Suggestion by Mining Click-Through and Session Data. In *Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*, pages 875–883, New York, NY, USA. ACM.

Caprile, C. and Tonella, P. (1999). Nomen Est Omen: Analyzing the Language of Function Identifiers. In *Proc. of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pages 112–122, Washington, DC, USA. IEEE Computer Society.

Card, S. K. and Nation, D. (2002). Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface. In *Proc. of the Working Conference on Advanced Visual Interfaces (AVI '02)*, pages 231–245, New York, NY, USA. ACM.

Chatterjee, S., Juvekar, S., and Sen, K. (2009). SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09) of the Joint European Conferences on Theory and Practice of Software (ETAPS '09)*, pages 385–400, Berlin, Heidelberg. Springer-Verlag.

Chirita, P.-A., Firan, C. S., and Nejdl, W. (2006). Summarizing Local Context to Personalize Global Web Search. In *Proc. of the 15th ACM International Conference on Information and Knowledge Management (CIKM '06)*, pages 287–296, New York, NY, USA. ACM.

Chirita, P. A., Nejdl, W., Paiu, R., and Kohlschütter, C. (2005). Using ODP Metadata to Personalize Search. In *Proc. of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '05)*, pages 178–185, New York, NY, USA. ACM.

Cohen, W. W., Schapire, R. E., and Singer, Y. (1999). Learning to Order Things. *Journal of Artificial Intelligence Research*, 10(1):243–270.

Collins, A. M. and Loftus, E. F. (1975). A Spreading-Activation Theory of Semantic Processing. *Psychological Review*, 82(6):407–428.

Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. (2005). Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465.

Daoud, M., Tamine-Lechani, L., Boughanem, M., and Chebaro, B. (2009). A Session Based Personalized Search Using an Ontological User Profile. In *Proc. of the 2009 ACM Symposium on Applied Computing (SAC '09)*, pages 1732–1736, New York, NY, USA. ACM.

de Alwis, B. and Murphy, G. C. (2008). Answering Conceptual Queries with Ferret. In *Proc. of the 30th International Conference on Software Engineering (ICSE '08)*, pages 21–30, New York, NY, USA. ACM.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. (2005). Towards Understanding Programs Through Wear-Based Filtering. In *Proc. of the ACM Symposium on Software Visualization (SoftVis '05)*, pages 183–192, New York, NY, USA. ACM.

Dey, A. K. and Abowd, G. D. (2000). Towards a Better Understanding of Context and Context-Awareness. In *Proc. of the CHI Workshop on the What, Who, Where, When, and How of Context-Awareness*, The Hague, The Netherlands.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271.

Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., and Sachs, J. (2004). Swoogle: A Search and Metadata Engine for the Semantic Web. In *Proc. of the 13th ACM International Conference on Information and Knowledge Management (CIKM 2004)*, pages 652–659, New York, NY, USA. ACM.

Dit, B., Guerrouj, L., Poshyvanyk, D., and Antoniol, G. (2011a). Can Better Identifier Splitting Techniques Help Feature Location? In *Proc. of the IEEE 19th International Conference on Program Comprehension (ICPC '11)*, pages 11–20, Washington, DC, USA. IEEE Computer Society.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2011b). Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software Maintenance and Evolution: Research and Practice*.

Doan, B.-L. and Brézillon, P. (2004). How the Notion of Context can be Useful to Search Tools. In *Proc. of the World Conference E-learn 2004*, Washington, DC, USA.

Dourish, P. (2004). What We Talk About When We Talk About Context. *Personal and Ubiquitous Computing*, 8(1):19–30.

Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K. (2009). Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Proc. of the 6th IEEE International Working Conference on Mining Software Repositories (MSR '09)*, pages 71–80, Washington, DC, USA. IEEE Computer Society.

Fellbaum, C., editor (1998). *WordNet: An Electronic Lexical Database.* MIT Press.

Ganter, B. and Wille, R. (1999). *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin/Heidelberg.

Gauch, S., Chaffee, J., and Pretschner, A. (2003). Ontology-Based Personalized Search and Browsing. *Web Intelli. and Agent Sys.*, 1(3-4):219–234.

Gay, G., Haiduc, S., Marcus, A., and Menzies, T. (2009). On the Use of Relevance Feedback in IR-Based Concept Location. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 351–360.

González, V. M. and Mark, G. (2004). "Constant, Constant, Multi-Tasking Craziness": Managing Multiple Working Spheres. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*, pages 113–120, New York, NY, USA. ACM.

Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (2005). *The Java Language Specification.* Addison-Wesley, Upper Saddle River, NJ, 3 edition.

Goth, G. (2005). Beware the March of this IDE: Eclipse is Overshadowing other Tool Technologies. *IEEE Software*, 22(4):108–111.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., and Cumby, C. (2010). A Search Engine for Finding Highly Relevant Applications. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 475–484, New York, NY, USA. ACM.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:199–220.

Guarino, N. (1998). Formal Ontology in Information Systems. In *Formal Ontology in Information Systems*, pages 3–15, Amsterdam. IOS Press.

Guerrouj, L., Di Penta, M., Antoniol, G., and Guéhéneuc, Y.-G. (2011). TIDIER: An Identifier Splitting Approach Using Speech Recognition Techniques. *Journal of Software Maintenance and Evolution: Research and Practice.*

Happel, H.-J. and Maalej, W. (2008). Potentials and Challenges of Recommendation Systems for Software Development. In *Proc. of the International Workshop on Recommendation Systems for Software Engineering (RSSE '08)*, pages 11–15, New York, NY, USA. ACM.

Harary, F., Norman, R. Z., and Cartwright, D. (1965). *Structural Models: An Introduction to the Theory of Directed Graphs.* Wiley, New York.

Harris, Z. (1954). Distributional Structure. *Word*, 10(23):146–162.

Heckmann, D., Schwartz, T., Brandherm, B., Schmitz, M., and von Wilamowitz-Moellendorff, M. (2005). GUMO - The General User Model Ontology. In *Proc. of the 10th International Conference on User Modeling (UM 2005)*, volume 3538 of *Lecture Notes in Computer Science*, pages 149–149. Springer Berlin/Heidelberg.

Heinemann, L., Bauer, V., Herrmannsdoerfer, M., and Hummel, B. (2012). Identifier-Based Context-Dependent API Method Recommendation. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 31–40, Szeged, Hungary. IEEE Computer Society.

Henninger, S. (1996). Supporting the Construction and Evolution of Component Repositories. In *Proc. of the 18th International Conference on Software Engineering (ICSE '96)*, pages 279–288, Washington, DC, USA. IEEE Computer Society.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating Collaborative Filtering Recommender Systems. *ACM Transactions on Information Systems*, 22(1):5–53.

Hill, E., Pollock, L., and Vijay-Shanker, K. (2007). Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 14–23, New York, NY, USA. ACM.

Hill, E., Pollock, L., and Vijay-Shanker, K. (2009). Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. of the IEEE 31st International Conference on Software Engineering (ICSE 2009)*, pages 232–242, Washington, DC, USA. IEEE Computer Society.

Holmes, R. and Murphy, G. C. (2005). Using Structural Context to Recommend Source Code Examples. In *Proc. of the 27th International Conference on Software Engineering (ICSE '05)*, pages 117–125, New York, NY, USA. ACM.

Hummel, O., Janjic, W., and Atkinson, C. (2008). Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software*, 25(5):45–52.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579.

Jannach, D., Zanker, M., Felfernig, A., and Friedrich, G. (2011). *Recommender Systems: An Introduction*. Cambridge University Press.

Janzen, D. and De Volder, K. (2003). Navigating and Querying Code Without Getting Lost. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 178–187, New York, NY, USA. ACM.

Jones, G. J. F. and Brown, P. J. (2004). The Role of Context in Information Retrieval. In *Proc. of the ACM SIGIR Workshop on Information Retrieval in Context*, Sheffield, UK.

Keivanloo, I., Roostapour, L., Schugerl, P., and Rilling, J. (2010). SE-CodeSearch: A Scalable Semantic Web-Based Source Code Search Infrastructure. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–5, Washington, DC, USA. IEEE Computer Society.

Kersten, M. and Murphy, G. C. (2005). Mylar: A Degree-of-Interest Model for IDEs. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 159–168, New York, NY, USA. ACM.

Kersten, M. and Murphy, G. C. (2006). Using Task Context to Improve Programmer Productivity. In *Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 1–11, New York, NY, USA. ACM.

Ko, A. J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. In *Proc. of the 29th International Conference on Software Engineering (ICSE '07)*, pages 344–353, Washington, DC, USA. IEEE Computer Society.

Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987.

Kotov, A., Bennett, P. N., White, R. W., Dumais, S. T., and Teevan, J. (2011). Modeling and Analysis of Cross-Session Search Tasks. In *Proc. of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11)*, pages 5–14, New York, NY, USA. ACM.

Krueger, C. W. (1992). Software Reuse. *ACM Comput. Surv.*, 24(2):131–183.

LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A. (2007). Program Comprehension as Fact Finding. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*, pages 361–370, New York, NY, USA. ACM.

Lawrie, D., Binkley, D., and Morrell, C. (2010). Normalizing Source Code Vocabulary. In *Proc. of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 3–12, Washington, DC, USA. IEEE Computer Society.

Lawrie, D., Morrell, C., Feild, H., and Binkley, D. (2006). What's in a Name? A Study of Identifiers. In *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 3–12, Washington, DC, USA. IEEE Computer Society.

Liu, F., Yu, C., and Meng, W. (2004). Personalized Web Search For Improving Retrieval Effectiveness. *IEEE Trans. on Knowl. and Data Eng.*, 16(1):28–40.

Lovins, J. B. (1968). Development of a Stemming Algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31.

Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. (2008). Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proc. of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 155–164.

Ma, Z., Pant, G., and Sheng, O. R. L. (2007). Interest-Based Personalized Search. *ACM Trans. Inf. Syst.*, 25(1).

Maedche, A. (2002). *Ontology Learning for the Semantic Web.* Kluwer Academic Publishers.

Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid Mining: Helping to Navigate the API Jungle. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 48–61, New York, NY, USA. ACM.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.

Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J. I. (2004). An Information Retrieval Approach to Concept Location in Source Code. In *Proc. of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223.

Matthijs, N. and Radlinski, F. (2011). Personalizing Web Search Using Long Term Browsing History. In *Proc. of the 4th ACM International Conference on Web Search and Data Mining (WSDM '11)*, pages 25–34, New York, NY, USA. ACM.

McCarey, F., Cinnéide, M. O., and Kushmerick, N. (2005). Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review*, 24(3-4):253–276.

McCarthy, J. and McCarthy, M. (2006). *Dynamics of Software Development.* Pro-Best Practices. Microsoft Press.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. (2011). Portfolio: Finding Relevant Functions and their Usage. In *Proc. of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 111–120, New York, NY, USA. ACM.

Mena, T. B., Saoud, N. B.-B., Ahmed, M. B., and Pavard, B. (2007). Towards a Methodology for Context Sensitive Systems Development. In *Proc. of the 6th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT 2007)*, volume 4635 of *Lecture Notes in Computer Science*, pages 56–68, Roskilde, Denmark. Springer.

Mihalkova, L. and Mooney, R. (2009). Learning to Disambiguate Search Queries from Short Sessions. In *Proc. of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part II (ECML PKDD '09)*, pages 111–127, Berlin, Heidelberg. Springer-Verlag.

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Mostefaoui, G. K., Pasquier-Rocha, J., and Brezillon, P. (2004). Context-Aware Computing: A Guide for the Pervasive Computing Community. In *Proc. of the IEEE/ACS International Conference on Pervasive Services (ICPS 2004)*, pages 39–48, Washington, DC, USA. IEEE Computer Society.

Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83.

Noy, N. F. and McGuinness, D. L. (2001). Ontology Development 101: A Guide to Creating Your First Ontology. Published: Stanford Knowledge Systems Laboratory Technical Report KSL-01-05.

Ogawa, Y., Morita, T., and Kobayashi, K. (1991). A Fuzzy Document Retrieval System Using the Keyword Connection Matrix and a Learning Method. *Fuzzy Sets Syst.*, 39(2):163–179.

Papagelis, M., Plexousakis, D., and Kutsuras, T. (2005). Alleviating the Sparsity Problem of Collaborative Filtering using Trust Inferences. In *Proc. of the 3rd International Conference on Trust Management (iTrust'05)*, pages 224–239, Berlin, Heidelberg. Springer-Verlag.

Parnin, C. and Gorg, C. (2006). Building Usage Contexts During Program Comprehension. In *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 13–22, Washington, DC, USA. IEEE Computer Society.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R., and Swart, C. (2012). Reactive Information Foraging: An Empirical Investigation of Theory-Based Recommender Systems for Programmers. In *Proc. of the ACM Annual Conference on Human Factors in Computing Systems (CHI '12)*, pages 1471–1480, New York, NY, USA. ACM.

Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., Burnett, M., and Bellamy, R. (2011). Modeling Programmer Navigation: A Head-to-Head Empirical Evaluation of Predictive Models. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 109–116, Pittsburgh, PA, USA. IEEE.

Pirolli, P. and Card, S. K. (1999). Information Foraging. *Psychological Review*, 106(4):643–675.

Pitkow, J., Schütze, H., Cass, T., Cooley, R., Turnbull, D., Edmonds, A., Adar, E., and Breuel, T. (2002). Personalized Search. *Commun. ACM*, 45(9):50–55.

Poshyvanyk, D. and Marcus, A. (2007). Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proc. of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 37–48.

Poshyvanyk, D., Marcus, A., and Dong, Y. (2006a). JIRiSS - An Eclipse Plug-in for Source Code Exploration. In *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 252–255.

Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D. (2006b). Source Code Exploration with Google. In *Proc. of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 334–338.

Ratanotayanon, S., Choi, H. J., and Sim, S. E. (2010). Using Transitive Changesets to Support Feature Location. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*, pages 341–344, New York, NY, USA. ACM.

Ratiu, D. and Deissenboeck, F. (2006). Programs are Knowledge Bases. In *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 79–83, Washington, DC, USA. IEEE Computer Society.

Resnick, P., Lacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proc. of the ACM Conference on Computer Supported Cooperative Work (CSCW '94)*, pages 175–186, New York, NY, USA. ACM.

Ribeiro, B. A. N. and Muntz, R. (1996). A Belief Network Model for IR. In *Proc. of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '96)*, pages 253–260, New York, NY, USA. ACM.

Robertson, S. E. and Jones, K. S. (1976). Relevance Weighting of Search Terms. *Journal of the American Society for Information Sciences*, 27(3):129–146.

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80 –86.

Robillard, M. P. (2008). Topology Analysis of Software Dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):18:1–18:36.

Robillard, M. P., Coelho, W., and Murphy, G. C. (2004). How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.*, 30(12):889–903.

Robillard, M. P. and Weigand-Warr, F. (2005). ConcernMapper: Simple View-Based Separation of Scattered Concerns. In *Proc. of the OOPSLA Workshop on Eclipse Technology eXchange*, pages 65–69, New York, NY, USA. ACM.

Rocchio, J. (1971). Relevance Feedback in Information Retrieval. In *The SMART Retrieval System*, pages 313–323. Prentice Hall.

Roget, P. M. (1852). *Roget's Thesaurus of English Words and Phrases.* Longman, London.

Sahavechaphan, N. and Claypool, K. (2006). XSnippet: Mining For Sample Code. In *Proc. of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pages 413–430, New York, NY, USA. ACM.

Salton, G. and Buckley, C. (1988). Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing and Management*, 24(5):513–523.

Salton, G., Fox, E. A., and Wu, H. (1983). Extended Boolean Information Retrieval. *Communications of the ACM*, 26(11):1022–1036.

Salton, G., Wong, A., and Yang, C. S. (1975). A Vector Space Model for Automatic Indexing. *Commun. ACM*, 18(11):613–620.

Saul, Z. M., Filkov, V., Devanbu, P., and Bird, C. (2007). Recommending Random Walks. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 15–24, New York, NY, USA. ACM.

Schein, A., Popescul, A., Ungar, L., and Pennock, D. (2002). Methods and Metrics for Cold-Start Recommendations. In *Proc. of the 25th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR '02)*, pages 253–260, New York, NY, USA. ACM.

Schilit, B. and Theimer, M. (1994). Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32.

Shao, P. and Smith, R. K. (2009). Feature Location by IR Modules and Call Graph. In *Proc. of the 47th Annual Southeast Regional Conference (ACM-SE 47)*, pages 70:1–70:4, New York, NY, USA. ACM.

Shen, X., Tan, B., and Zhai, C. (2005). Context-Sensitive Information Retrieval Using Implicit Feedback. In *Proc. of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '05)*, pages 43–50, New York, NY, USA. ACM.

Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., and Vijay-Shanker, K. (2007). Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. of the 6th International Conference on Aspect-Oriented Software Development (AOSD '07)*, pages 212–224, New York, NY, USA. ACM.

Sillito, J., Murphy, G. C., and De Volder, K. (2006). Questions Programmers Ask During Software Evolution Tasks. In *Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 23–34, New York, NY, USA. ACM.

Sillito, J., Murphy, G. C., and Volder, K. D. (2008). Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451.

Singer, J., Elves, R., and Storey, M.-A. (2005). NavTracks: Supporting Navigation in Software Maintenance. In *Proc. of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, Budapest, Hungary. IEEE.

Smullyan, R. M. (1968). *First-Order logic.* Springer-Verlag.

Sontag, D., Collins-Thompson, K., Bennett, P. N., White, R. W., Dumais, S., and Billerbeck, B. (2012). Probabilistic Models for Personalizing Web Search. In *Proc. of the 5th ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 433–442, New York, NY, USA. ACM.

Speretta, M. and Gauch, S. (2005). Personalized Search Based on User Search Histories. In *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence (WI '05)*, pages 622–628, Washington, DC, USA. IEEE Computer Society.

Staab, S. and Maedche, A. (2001). Knowledge Portals: Ontologies at Work. *AI Magazine*, 22:63–75.

Starke, J., Luce, C., and Sillito, J. (2009). Searching and Skimming: An Exploratory Study. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 157–166, Edmonton, Alberta, Canada. IEEE.

Stuckenschmidt, H. and van Harmelen, F. (2005). *Information Sharing on the Semantic Web.* Advanced Information and Knowledge Processing. Springer.

Sugiyama, K., Hatano, K., and Yoshikawa, M. (2004). Adaptive Web Search Based on User Profile Constructed without Any Effort from Users. In *Proc. of the 13th International Conference on World Wide Web (WWW '04)*, pages 675–684, New York, NY, USA. ACM.

Takang, A. A., Grubb, P. A., and Macredie, R. D. (1996). The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. *J. Prog. Lang.*, 4(3):143–167.

Tan, B., Shen, X., and Zhai, C. (2006). Mining Long-Term Search History to Improve Search Accuracy. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*, pages 718–723, New York, NY, USA. ACM.

Teevan, J., Dumais, S. T., and Horvitz, E. (2005). Personalizing Search via Automated Analysis of Interests and Activities. In *Proc. of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '05)*, pages 449–456, New York, NY, USA. ACM.

Thummalapenta, S. and Xie, T. (2007). PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 204–213, New York, NY, USA. ACM.

Turtle, H. and Croft, W. B. (1990). Inference Networks for Document Retrieval. In *Proc. of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '90)*, pages 1–24, New York, NY, USA. ACM.

Turtle, H. and Croft, W. B. (1991). Evaluation of an Inference Network-Based Retrieval Model. *ACM Trans. Inf. Syst.*, 9(3):187–222.

van Heijst, G., Schreiber, A., and Wielinga, B. (1997). Using Explicit Ontologies for KBS Development. *International Journal of Human-Computer Studies*, 42:183–292.

von Mayrhauser, A. and Vans, A. M. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55.

Warr, F. W. and Robillard, M. P. (2007). Suade: Topology-Based Searches for Software Investigation. In *Proc. of the 29th International Conference on Software Engineering (ICSE '07)*, pages 780–783, Washington, DC, USA. IEEE Computer Society.

Wiese, A., Ho, V., and Hill, E. (2011). A Comparison of Stemmers on Source Code Identifiers for Software Search. In *Proc. of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, pages 496–499, Washington, DC, USA. IEEE Computer Society.

Wilkinson, R. and Hingston, P. (1991). Using the Cosine Measure in a Neural Network for Document Retrieval. In *Proc. of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '91)*, pages 202–210, New York, NY, USA. ACM.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edition.

Wong, S. K. M., Ziarko, W., and Wong, P. C. N. (1985). Generalized Vector Space Model in Information Retrieval. In *Proc. of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '85)*, pages 18–25, New York, NY, USA. ACM.

Wursch, M., Ghezzi, G., Reif, G., and Gall, H. C. (2010). Supporting Developers with Natural Language Queries. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 165–174, New York, NY, USA. ACM.

Xiang, B., Jiang, D., Pei, J., Sun, X., Chen, E., and Li, H. (2010). Context-Aware Ranking in Web Search. In *Proc. of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '10)*, pages 451–458, New York, NY, USA. ACM.

Ye, Y. and Fischer, G. (2002). Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *Proc. of the 24th International Conference on Software Engineering (ICSE '02)*, pages 513–523, New York, NY, USA. ACM.

Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. (2004). Predicting Source Code Changes by Mining Change History. *IEEE Trans. Softw. Eng.*, 30(9):574–586.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). MAPO: Mining and Recommending API Usage Patterns. In *Proc. of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, Genoa, pages 318–343, Berlin, Heidelberg. Springer-Verlag.

Zimmermann, A., Lorenz, A., and Oppermann, R. (2007). An Operational Definition of Context. In *Proc. of the 6th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT 2007)*, volume 4635 of *Lecture Notes in Computer Science*, pages 558–571, Roskilde, Denmark. Springer.

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

Zuniga, G. L. (2001). Ontology: Its Transformation From Philosophy to Information Systems. In *Proc. of the International Conference on Formal Ontology in Information Systems*, pages 187–197. ACM Press.