

Developer-Centric Models: Easing Access to Relevant Information in a Software Development Environment

by

Thomas Fritz

Dipl. Informatiker, Ludwig-Maximilians-Universität München, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

April 2011

© Thomas Fritz, 2011

Abstract

During the development of a software system, large amounts of new information, such as source code, work items and documentation, are produced continuously. As a developer works, one of his major activities is to consult portions of this information pertinent to his work to answer the questions he has about the system and its development. Current development environments are centered around models of the artifacts used in development, rather than of the people who perform the work, making it difficult and sometimes infeasible for the developer to satisfy his information needs.

We introduce two developer-centric models, the degree-of-knowledge (DOK) model and the information fragments model, which support developers in accessing the small portions of information needed to answer the questions they have. The degree-of-knowledge model computes automatically, for each source code element in the development environment, a real value that represents a developer's knowledge of that element based on a developer's authorship and interaction data. We present evidence that shows that both authorship and interaction information are important in characterizing a developer's knowledge of code. We report on the usage of our model in case studies on expert finding, knowledge transfer and identifying changes of interest. We show that our model improves upon an existing expertise finding approach and can accurately identify changes for which a developer should likely be aware. Finally, we discuss the robustness of the model across multiple development sites and teams.

The information fragment model automates the composition of different kinds of information and allows developers to easily choose how to display the composed information. We show that the model supports answering 78 questions that involve

the integration of information siloed by existing programming environments. We identified these questions from interviews with developers. We also describe how 18 professional developers were able to use a prototype tool based on our model to successfully and quickly answer 94% of eight of the 78 questions posed in a case study. The separation of composition and presentation supported by the model, allowed the developers to answer the questions according to their personal preferences.

Preface

The research in this thesis has been previously published in the following articles:

1. “Does a Programmer’s Activity Indicate Knowledge of Code?” T. Fritz, G. C. Murphy and E. Hill. In *ESEC-FSE’07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIG-SOFT symposium on The foundations of software engineering*, pages 341–350, 2007. ACM.
2. “A Degree-of-Knowledge Model to Capture Source Code Familiarity.” T. Fritz, J. Ou, G. C. Murphy and E. Murphy-Hill. In *ICSE’10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 385–394, 2010. ACM.
3. “Using Information Fragments to Answer the Questions Developers Ask.” T. Fritz and G. C. Murphy. In *ICSE’10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184, 2010. ACM.
4. “Staying Aware of Relevant Feeds in Context.” T. Fritz. In *ICSE’10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 523–524, 2010. ACM.

All four papers are under the copyright of the ACM and for which I have permission to reuse.

Part of this work involved other collaborators. In particular Jingwen Ou, Emerson Murphy-Hill and Emily Hill. Jingwen Ou participated in the implementation and the collection of the authorship and interaction data used in the degree-of-knowledge model. Emerson Murphy-Hill and Emily Hill contributed comments and figures to articles mentioned above that also partially flowed into this dissertation. Emily Hill also helped with the statistical analysis of the data collected for

the study I conducted to determine whether a programmer's activity can indicate knowledge of code.

The UBC Behavioural Research Ethics Board approved the research in the certificate H06-03693, "An Investigation of Knowledge Acquisition During Software Development" and in amendments and renewals to this certificate (H06-03693-A001, H06-03693-A002, H06-03693-A003, H06-03693-A004, H06-03693-A005, H06-03693-A006 and H06-03693-A007).

Table of Contents

Abstract	ii
Preface	iv
Table of Contents	vi
List of Tables	x
List of Figures	xii
Acknowledgments	xiii
1 Introduction	1
1.1 A Model of a Developer's Knowledge of Code	3
1.1.1 Earlier Efforts	3
1.1.2 The Degree-of-Knowledge Model	4
1.1.3 Evaluating the Model	5
1.2 A Model to Integrate Multiple Kinds of Information	6
1.2.1 Earlier Efforts	7
1.2.2 The Information Fragments Model	8
1.2.3 Evaluating the Model	8
1.3 Contributions	9
1.4 Organization	10
2 Related Work	11
2.1 Modeling a Developer's Knowledge	12
2.1.1 Expertise Recommenders Based on Authorship	12

2.1.2	Using Interaction to Support Developers	13
2.1.3	Explicitly Modeling a Developer's Knowledge	14
2.2	Modeling a Developer's Information Needs Across Multiple Kinds of Information	15
2.2.1	Studies of Developers' Questions	16
2.2.2	Answering a Developer's Questions	17
3	The Degree-of-Knowledge Model	22
3.1	A Developer's Activity as an Indicator for Knowledge	23
3.1.1	Study	23
3.1.2	Quantitative Results	29
3.1.3	Qualitative Results	32
3.1.4	Threats to Validity	37
3.1.5	The Need for a Better Indicator of a Developer's Knowledge	38
3.2	Authorship and Interaction	38
3.2.1	Authorship—a Long-term Component	40
3.2.2	Interaction—a Short-term Component	43
3.2.3	Authorship and Interaction	46
3.3	Degree-of-Knowledge Model	48
3.3.1	Degree-of-Authorship	48
3.3.2	Degree-of-Interest	48
3.3.3	Degree-of-Knowledge	48
3.4	Determining DOK Weightings	49
3.4.1	Method	49
3.4.2	Analysis and Results	50
4	Evaluation of the DOK Model	53
4.1	Case Studies	53
4.1.1	Finding Experts	54
4.1.2	Onboarding	59
4.1.3	Identifying Changes of Interest	61
4.1.4	Case Studies Summary	63
4.2	Robustness of the Model	63

4.2.1	Differences in Development Teams	63
4.2.2	One Model for All	66
4.2.3	A Site-Specific Model	68
4.2.4	Robustness Summary	69
4.3	Threats to Validity	70
4.3.1	Amount of Data	70
4.3.2	Multiple Stream Development	70
4.3.3	Project Phase	70
4.3.4	Individual Factors	71
5	The Information Fragments Model	72
5.1	Developers' Questions	72
5.1.1	Subjects and Interview Process	73
5.1.2	Interview Results	73
5.1.3	Threats	79
5.2	Answering Questions Using Existing Approaches	79
5.2.1	Using an Integrated Development Environment	79
5.2.2	Using a Query Language	80
5.3	Information Fragment Model	81
5.3.1	Example of Use	82
5.3.2	Information Fragments	84
5.3.3	Composition Operators	85
5.3.4	Presentation	86
6	Evaluation of the Information Fragments Model	88
6.1	Applying the Model	88
6.2	Prototype	99
6.2.1	Information Fragments	99
6.2.2	Composition	99
6.2.3	Presentation	100
6.2.4	Completeness of the Prototype	101
6.3	Study	102
6.3.1	Subjects	103

6.3.2	Study Method	103
6.3.3	Data Analysis	107
6.3.4	Can Developers Use the Model?	107
6.3.5	How Do Developers Use the Model?	110
6.3.6	What Do Developers Think About the Approach?	117
6.4	Threats	119
7	Discussion and Future Work	121
7.1	Degree-of-Knowledge Model	121
7.1.1	Structural Knowledge vs. User Rating	122
7.1.2	Linear Regression	122
7.1.3	API Elements, Structural Information and Usage Expertise	122
7.1.4	Using DOK to Prevent Bugs	123
7.1.5	Finding Relevant Bugs	123
7.1.6	Longitudinal Study	124
7.2	Information Fragments Model	124
7.2.1	Automatic Composition	124
7.2.2	Text Matching and Other Operators	125
7.2.3	Model vs SQL	126
7.2.4	Information Fragment Selection	126
7.2.5	Extending the Tool	127
7.2.6	Presenting the Information	127
7.3	Using Knowledge and Context for Awareness	128
8	Conclusion	131
	Bibliography	134
	Appendix A Studies: Supporting Materials	144
A.1	Sample Questions for Study _{EXP_DOK}	144
A.2	Sample Questions for Study _{INFR}	145
A.3	Sample Questions for Study _{FEEDS}	146
A.4	Tutorial for the Evaluation of the Information Fragments Model	146

List of Tables

3.1	Monitored Interaction Events (see [52] for more detail)	25
3.2	General Interview Questions	33
3.3	Average of Six Weeks of a Developer's Authorship Data Over a Three Month Period (FA = First Authored, DL = Delivered, AC = Accepted, Change stands for the change in elements per week) . . .	41
3.4	Developer's Interaction Data Averaged Over Five Work Days	44
3.5	Coefficients for Linear Regression	51
4.1	Size of Bug Recommendation Sets	62
4.2	Average of Six Weeks of a Developer's Authorship Data Over a Three Month Period (FA = First Authored, DL = Delivered, AC = Accepted, Change stands for the change in elements per week; subjects C1 to C5 are from Site ₂ , subjects E1 and E2 from Site ₃) . .	65
4.3	Developer's Interaction Data Averaged Over Five Work Days (sub- jects C1 to C5 are from Site ₂ , subjects E1 and E2 from Site ₃)	66
4.4	Coefficients for Linear Regression over all Sites.	67
4.5	Coefficients for Linear Regression for Site ₂	69
5.1	Developers' Questions and the Operators and Domains for Desired Answers (*: <i>question explicitly stated by a developer</i> , <i>id</i> : <i>identifier</i> <i>matching</i> , <i>t</i> : <i>text matching</i>)	75
5.2	Sample Node Domains, Types & Properties	85
6.1	Information Fragments For Answering Developer's Questions . . .	90
6.2	Using the Model to Answer all 78 Questions.	93
6.3	Study Questions (original developer question in brackets)	105

6.4	The Ten Base Fragments Available to Participants	105
6.5	Developer's Results	108
6.6	Answer Variations (<i>hidden</i> refers to the case in which a developer used the hide action to elide an information fragment from the pre- sentation)	112
6.7	Correlations Between Time (T) and Fragments Used (FU), Re- orderings (RO) and Restarts (RS)	116
A.1	Studies Conducted for this Thesis	145

List of Figures

1.1	Answering “What Have My Coworkers Been Doing?” with the Information Fragment Model.	9
3.1	Types of Questions in the Questionnaire	26
3.2	Questionnaire with Open Type Action	28
3.3	Correct Answers by Subject for Questionnaire 1, 2 and 3	30
3.4	Data Collection Time Periods	40
3.5	Positive DOI Elements	45
3.6	Authorship and Interaction over Five Days	47
4.1	Part of a Knowledge Map	56
5.1	Approach to Answer the Question “What have people been working on?”	83
6.1	Views of the Prototype	101
6.2	Pre-defined Base Fragments in the Prototype	106
6.3	Variations in Answer over 18 Participants (bars represent number of participants that used one particular solution to a question; σ^* : count functionality that allows you to count children of an element in the tree)	111
6.4	Aspects in the Process of Answering Questions	114
6.4	Aspects in the Process of Answering Questions (continued)	115
7.1	Presentation of Feeds.	129

Acknowledgments

The mediocre teacher tells. The good teacher explains. The superior teacher demonstrates. The great teacher inspires. – William A. Ward

My deepest thanks go to a great teacher, my supervisor Gail Murphy. Gail inspired me to do research and to do a PhD. She not only taught me how to do research and how fun research can be, she is better described as my “doctor mother”—the German terminology for a PhD supervisor. I cannot count the times I dropped by her office, asked her for advice, chatted with her and yet, she always had an open door, a smile and encouragement for me. **Thank you!**

I also want to thank Gregor Kiczales and David Poole on my supervisory committee for the advice and support they provided to me.

There have been a lot of people along the way who inspired and supported me in starting and completing my thesis and I want to thank all of them.

Special thanks go to my parents Edi and Angelika who inspired me to follow my dreams and with their love and support contributed tremendously to who I am and where I am today. Special thanks also go to my two brothers Alex and Olli and their partners Susi and Petra. It is amazing to have them in my life and I can always count on them.

I would also like to thank all my friends who supported me, taught me a lot and made my life a lot happier and brighter. I cannot express how grateful I am for all you have done for me. In alphabetical order: Bram, Brett, Brian, Cam, Chris, Claire, Clint, Davey, several Davids, Denise, Derek, Ellen, Emerson, Emily, Greg, Jelena, several Jenns, Julia, Julian, Laura, Liz, Mandeep, Meghan, Pooja, Rainer, Ryan, Sarah, Scott, Shawn, Stefan, Stephi, Terry, Thomas. Thanks to all the people

in SPL, the triathlon and the sailing club and the rowing team at UBC.

I am also thankful for the many people at IBM who made all my studies possible, in particular Marcellus, Maria, Jean-Michel and Sara, and I am thankful for the support of the IBM Center for Advanced Studies in general, the support of the Natural Sciences and Engineering Research Council of Canada, the University of British Columbia and Ms. Sudbury.

Chapter 1

Introduction

One of the effects of living with electric information is that we live habitually in a state of information overload. There's always more than you can cope with. – Marshall McLuhan

In the development of a software system, large amounts of new information are produced continuously. Source code, bugs, iteration plans and documentation, to name just a few, are changed or newly created by developers of the software system every day. As one example of the voluminous amounts of information generated, over the approximately 15 months of one development cycle of Eclipse—an open-source integrated development environment—around 10 thousand Java source files were created or changed, more than 26 thousand Bugzilla bugs were entered and more than 45 thousand newsgroup entries were posted [1].

As part of producing this new information, a developer must continuously answer questions about the current state of the project [76, 78]. Answering these questions typically requires searches to be performed over the large amounts of system information to find the small portion that is pertinent to the developer's work. One of the commonly asked questions Ko and colleagues found by observing seventeen professional developers at Microsoft is “What have my coworkers been doing?” [55]. Answering this question requires finding just the changes completed by members of one's team amongst the many changes that may have been completed by the entire software development staff for the project. Given

the long history of integrated development environments (IDEs) used by software developers and the many capabilities in these environments, one would expect that answering questions of this nature would be easy.

Unfortunately, answering such questions is not straightforward because today's development environments are centered around models of the artifacts used in development, rather than of the people who perform the work. For instance, a central model in a development environment is an abstract syntax tree [38], which provides an abstraction of source code to facilitate feedback about the syntax of code being written, facilitate navigation in terms of code structure and facilitate code transformations. By providing feedback about artifacts, these models benefit the developers using the environment. However, these models fall short for developers by supporting only a small fraction of their information needs when they are performing work on the system. In particular, instead of helping a developer pinpoint just that information needed to perform work at the moment, development environments have been designed and engineered to provide information about all artifacts associated with the system. The result for the developer is that it is often difficult, and sometimes even infeasible, to find the answer to a question of interest.

Our thesis is that *developer-centric models can be combined with artifact-centric models in a development environment to ease a developer's access to the information relevant to the work-at-hand*. Specifically, we introduce two developer-centric models to support a developer in answering a broader set of questions related to his information needs than is possible with existing research. One model, the *degree-of-knowledge model*, represents a developer's knowledge of the source code comprising a system of interest. This model can be used to identify information about the system in which the developer might be interested and who the developer can ask to find out about parts in which the developer lacks knowledge. The second model, the *information fragments model*, provides a means for a developer to integrate and compose information about the system to answer questions that are infeasible to answer with current approaches. We now proceed to describe each of these models in more detail.

1.1 A Model of a Developer’s Knowledge of Code

On large system developments, software developers tend to work in specific parts of the system’s code base. For instance, one team may be responsible for the web-based user interface to a system whereas another team on the development project may be responsible for efficient storage of server-side data. When a member of the first team needs to understand the code about how data is stored on the server, the developer needs to be able to identify which member of the server-side data team is the appropriate person to ask. This situation of needing to know *who to ask* arises frequently on projects [56, 60]. In addition to determining who to ask, developers also need to stay aware of changes that might break their code [18, 57], asking *which changes should I know about*.

Today’s development environments model only the artifacts and not the individual developer’s perspectives of a system that is needed to support answering questions specific to a developer’s knowledge. Existing research to support answering these types of questions also lacks focus on the individual developer, making it difficult to answer such developer-specific questions. A model that approximates an individual developer’s knowledge of code can be used to better support these questions.

1.1.1 Earlier Efforts

Researchers have proposed support for answering the question of *who to ask* through recommenders that suggest experts for parts of the code based on the authorship of changes to the code (e.g., [60, 63]). A tacit assumption with these existing recommenders is that a developer’s changes to the code indicate his knowledge about the code. These approaches focus on a system-wide knowledge model for code, trying to find one expert for each code element. By focusing on the code instead of the developer, these approaches fall short by assuming that authorship is the only way that a developer can gain knowledge about code. In fact, developers must interact with the code prior to authoring the code. These approaches also treat a developer’s knowledge as a monotonically increasing function whereas, in reality, a developer’s knowledge ebbs and flows as different developers change the same part of the code base.

Researchers have also previously considered the question of *which change should a developer know about* by flowing all notifications on changes to developers through such mechanisms as mailing lists and commit logs [37] or by displaying the changes directly in the developer’s environment [7, 71, 72]. These approaches place the burden on the developer to identify changes of interest. Given how much information can change in a software system within a short amount of time, this identification can be time-consuming and tedious [37].

Researchers have also more generally studied how to explicitly model the different parts of a developer’s knowledge and how developers comprehend code (e.g., [9, 54, 82]). These approaches focus on the mechanisms of gaining knowledge rather than what knowledge a developer retains as a result.

1.1.2 The Degree-of-Knowledge Model

To address limitations in these earlier efforts, we introduce the degree-of-knowledge (DOK) model that approximates an individual developer’s knowledge of the system’s code. Similar to artifact-centric models, the DOK model can be embedded within a development environment and can be determined automatically.

As an initial step towards a model, we performed an exploratory study with 19 professional software developers (Study_{EXP-DOK}) to investigate the factors that should be used in modeling a developer’s knowledge (Section 3.1). In this study, we investigated a developer’s interaction with the code as a proxy for his knowledge, since interactions always precede and are part of authoring changes. The study establishes, with statistical significance, that the more frequently and recently a developer interacts with a code element (as represented by a high degree of interest (DOI) value [52]), the more the developer knows about the element. From interviews with 13 of the 19 developers, we also determined a number of other factors that may be used to model a developer’s knowledge, such as the authorship of program elements and the code stability.

Given our pursuit to define a model that can determine automatically a developer’s knowledge of code, we chose to focus on the interaction and authorship aspects. We gathered data in another study, Study_{DATA-DOK}, from seven professional developers at one development site, which we refer to as Site₁. We report on this

data to confirm two hypotheses (Section 3.2). First, the code that developers work on changes rapidly. Second, authorship and interaction each capture a unique and valuable perspective on a developer’s knowledge. Authorship represents a longer-term component of a developer’s knowledge; interaction indicates a shorter-term component.

Using the results of Study_{EXP_DOK} and Study_{DATA_DOK} , we developed the degree-of-knowledge (DOK) model to capture a developer’s individual knowledge of code (Section 3.3). The DOK model assigns a real value to each code element in a developer’s development environment and is based on a linear combination of a developer’s interaction with an element, computed by the DOI, and three factors of authorship: whether the developer was the first author of the element (FA), the number of changes the developer has contributed to the element (DL) and the number of changes others have made to the element (accepted changes – AC).

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI$$

This combination accounts for knowledge gained by a developer interacting with the code for such purposes as trying to understand how the code functions as well as the knowledge that results from creating the code. This model also accounts for the ebb and flow of a developer’s knowledge; for instance, a developer’s knowledge of a code element decreases when someone else changes the element. The DOK model enables the automatic computation of the individual knowledge of each developer in a source code element by combining authorship data from the source revision system and interaction data from monitoring the developer’s activity in the development environment.

To determine the weighting factors for each of the four factors (FA , DL , AC , DOI), we conducted an experiment, Experiment_{DOK} , with the seven developers from Site_1 (Section 3.4). We found that both authorship and interaction improve the quality of the model and help to explain a developer’s knowledge of an element.

1.1.3 Evaluating the Model

The availability of an individual DOK model for each developer in a team opens up several possibilities to improve a developer’s productivity and quality of work. We consider three possibilities in this thesis through exploratory case studies (Study_{CS_DOK})

that we conducted with three different teams at three different sites (Section 4.1). These studies consider three questions that were raised by others as important in the literature: who should I ask about a part of the system’s code base [56, 60], which changes to the code base should I know about [25, 44] and what code do I need to know about [18, 57]. We found that the DOK model performed better than existing approaches for finding the expert in parts of a code base across teams with different code ownership styles. We also found that the DOK model can help to accurately identify changes of which a developer should likely be aware and learned about kinds of source code for which our current definition of DOK does not adequately reflect a developer’s knowledge.

To examine whether the DOK model is of value in other environments, such as different teams, different project phases or different working styles, we collected data from two more teams from different sites (Study_{DATA.2.DOK}). We found that despite their significance, differences in the authorship and interaction behavior between the teams only have a minor impact on the value of the DOK model.

1.2 A Model to Integrate Multiple Kinds of Information

The degree-of-knowledge developer-centric model provides a developer’s perspective on one kind of information in the development environment, namely source code. A developer also has information needs when working on the system that span across multiple kinds of information.

Consider the question “What have my coworkers been doing?” [55] again. Depending upon the developer asking the question, an answer may involve only information from a revision system to determine what code is changing or it may also involve information from a work item repository¹ to help explain why the change is occurring. While earlier work points at the need for answering questions across multiple kinds of information [55, 57], this work states questions at an abstract level and does not discuss the ambiguity that lies in the developer’s interpretations of these questions.

To better understand the range of questions of this form that are asked by developers and to better understand the concrete forms of the questions, we performed

¹Work items are similar to bug reports, issues or tasks.

an exploratory study, *Study_{EXP_FR}*. We interviewed eleven professional software developers and identified 78 questions of interest to them that have the characteristic of requiring multiple kinds of information (Section 5.1). We also found that even though a lot of the questions sound similar, there can be substantial variation in how a developer interprets a question.

With existing development environments that put different kinds of information into different silos [21], answering this type of questions is difficult. To answer the question “What have my coworkers been doing?” a developer has to follow links from work items to change sets to source code, with each link navigation resulting in information displayed in another view. The burden is placed on the developer to manually and mentally correlate the information between the views to answer the question of interest. Existing research to support this type of questions is either too expressive (e.g., [53]) or too limited (e.g., [81]) in the way information can be integrated, making it difficult or respectively infeasible to answer such questions. A model that matches more closely a developer’s needs for integrating information can be used to better support these questions.

1.2.1 Earlier Efforts

One previous approach to help developers answer such questions is to provide a query language supporting queries across all the kinds of information involved (e.g., [53, 67]). In this case, the developer must know or learn the query language and must specify explicitly how the different kinds of information should be integrated. To answer “What have my coworkers been doing?”, the developer needs to specify how work item, change set and source code information should be integrated (see Section 5.2 for more detail).

Another approach to support such questions is to provide a fixed integration of different kinds of information through such means as a fixed schema (e.g., [81]), an explicit ontology that models different kinds of information and relations between the kinds (e.g., [47]) or by overlaying the information in a view (e.g., [26]). These approaches restrict the way information can be integrated and thus limit the flexibility required to answer the multitude of questions.

1.2.2 The Information Fragments Model

To enable developers to answer questions that require multiple kinds of information, we introduce the information fragment model that supports the automatic integration of different kinds of information using the structure of the information (Section 5.3). A developer can indicate which portions of information in the development environment to integrate, which we refer to as *information fragments*, and can adapt the *presentation* of resultant integrated information to support answering the questions according to personal preferences. The integration of the information, which we refer to as *composition* is done automatically.

To answer the question “What have my coworkers been doing?” with the model, a developer can select four information fragments: one comprising the team on which he works, one comprising the work items worked on over a selected time period, one comprising change sets over a selected time period, and one comprising the source code in his workspace. The developer then drags and drops these fragments into a special view that supports the model and orders the presentation of the fragments according to his preference. Figure 1.1 presents the result of an automatic composition, where the presentation is based on team members first followed by work items, change sets and source code. This view makes it easy to see that Alex worked on two work items over the time period of interest and to see the changes to the code performed for these work items.

1.2.3 Evaluating the Model

Using our model, we have been able to express all 78 questions determined through the interview study Study_{EXP.FR} (Section 5.1). With a prototype tool we implemented to support the model (Section 6.2), we conducted a case study Study_{INFR}. In the study, 18 professional developers used the prototype tool to answer eight of the 78 questions that span across multiple kinds of information. We found that developers were able to easily apply the model to successfully answer 94% of eight questions posed. We also found that developers used the model in different ways to answer the same question, suggesting that the approach supports individual preferences (Section 6.3).

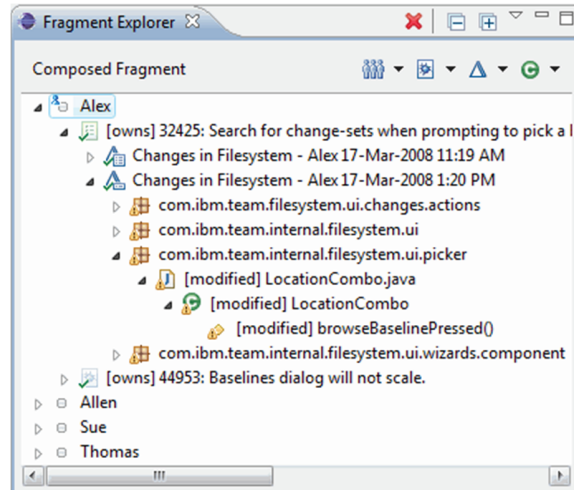


Figure 1.1: Answering “What Have My Coworkers Been Doing?” with the Information Fragment Model.

1.3 Contributions

This thesis makes contributions with respect to two developer-centric models, the degree-of-knowledge model that represents a developer’s knowledge of code, and the information fragments model that provides a means for a developer to integrate information about the system according to the developer’s personal preferences. For the degree-of-knowledge model:

- we show empirically that a developer’s interaction with code can indicate his knowledge and enumerate additional factors that affect a developer’s knowledge of code;
- we present evidence that shows a developer’s authorship of and interaction with code capture different aspects of knowledge about code;
- we introduce the degree-of-knowledge (DOK) model, an individual view of a developer’s knowledge of code;
- we report on the use of the DOK model in three scenarios, discussing benefits and limitations of the model; and

- we discuss the robustness of the model across a variety of development teams.

For the information fragment model:

- we identified and present 78 questions that developers ask and how the developers desire these questions to be answered;
- we introduce the information fragment model and show that it is sufficiently expressive to answer all 78 questions;
- we demonstrate that developers can successfully apply the model through a study with 18 professional developers.

1.4 Organization

In Chapter 2 we present earlier research efforts related to the problems addressed in this thesis. In Chapter 3, we describe the degree-of-knowledge model together with the results from the studies to determine the factors of the model and their relative effect. In Chapter 4, we present our evaluation of the degree-of-knowledge model based on three scenarios we performed at different development sites to explore the use of DOK models and a discussion of the DOK model's applicability in different environments. Our second model, the information fragments model, is presented in Chapter 5, together with the exploratory study we performed to determine the range of questions across multiple kinds of information. In Chapter 6, we present an evaluation of the model, by, first, showing how all 78 questions determined through the interview study can be expressed using the model and, second, in form of a case study with professional developers. In Chapter 7, we discuss aspects and future work of each of the two models and describe how the two models can be synergistically used together to support a developer's project awareness. Finally, we conclude and summarize the thesis in Chapter 8.

Chapter 2

Related Work

For many years, to help manage the complexity of building software systems, software developers have built and relied upon the services of integrated development environments. Early development environments, such as Interlisp [11, 80], provided developers with information about how the software they were constructing connected together, such as which functions called which other functions. Later efforts, expanded the capabilities of development environments beyond easing navigation across source code to include such concepts as configuration management (e.g., [12]). As building software development environments with significant functionality is expensive and often tailored to one or a small number of programming languages, researchers have also considered how to generate these environments (e.g., [38]).

This previous research has resulted in development environments that provide significant aid to software developers. However, as discussed in Chapter 1, these development environments are centered around models for the artifacts, making it difficult or infeasible to answer questions that address a developer’s individual information needs. The two models we introduce in this thesis increase the information needs of a developer that can be satisfied within a development environment.

The degree-of-knowledge (DOK) model we introduce seeks to support developers by modeling a developer’s knowledge. We begin our discussion of earlier efforts in Section 2.1 with a review of approaches that have either implicitly or explicitly modeled a developer’s knowledge of code. The information fragments

model seeks to support developers in answering questions across multiple kinds of information by integrating information according to the developer’s needs. In Section 2.2 we first discuss empirical studies on questions across multiple kinds of information, before reviewing approaches that support developers in answering this type of questions by integrating information in an either loose or fixed manner.

2.1 Modeling a Developer’s Knowledge

Previous research on modeling a developer’s knowledge of code can be categorized as taking either an implicit or an explicit approach. Implicit approaches make tacit assumptions on how data retrieved from a developer can be used to model what the developer knows or wants to know. These implicit approaches can further be categorized into approaches that use authorship data (Section 2.1.1) or interaction data (Section 2.1.2) to support developers. Explicit approaches empirically study programmers, resulting in models of how to describe different parts of knowledge, how developers comprehend code or the type of knowledge programmers have (Section 2.1.3).

2.1.1 Expertise Recommenders Based on Authorship

Previous automated approaches to determining the familiarity (expertise) of developers with a code base rely solely on change information. For instance, the Expertise Recommender [60] and Expertise Browser [63] each use a form of the “Line 10 Rule”, which is a heuristic that the person committing changes to a file¹ has expertise in that file. The Expertise Recommender uses this heuristic to present the developer with the most recent expertise for the source file; the Expertise Browser gathers and ranks developers based on changes over time. The Emergent Expertise Locator refines the approach of the Expertise Browser by considering the relationship between files that were changed together when determining expertise [62]. Girba and colleagues consider finer-grained information, equating expertise with the number of lines of code each developer changes [35]. Hattori and colleagues consider changes that have not yet been committed [41]. These systems are able

¹We use the term file, but many of these techniques also apply at a finer-level of granularity, such as methods or functions.

to recommend with a level of precision and recall that appears useful to developers in the limited studies conducted. We may be able to improve the effectiveness of these kinds of tools if we develop a better understanding of what kinds of activity lead to particular kinds of knowledge.

None of these previous approaches consider the ebb and flow of a developer's expertise in a particular part of the system. The Expertise Recommender considers expertise as a binary function, only one developer at a time has expertise in a file depending on who last changed it. The Expertise Browser and Emergent Expertise Locator represent expertise as a monotonically increasing function; a developer who completely replaces the implementation of an existing method has no impact on the expertise of the developer who originally created the method. Our DOK models the ebb and flow of multiple developers changing the same file; a developer's degree-of-knowledge in the file rises when the developer commits changes to the source repository and diminishes when other developers make changes.

The DOK model we develop also differs from previous expertise identification approaches by considering not just the code a developer authors and changes, but also code that the developer consults during their work. Schuler and Zimmermann also noted the need to move beyond authorship for determining expertise, suggesting an approach that analyzed the changed code for what code was called (but not changed) [73]. In this way, they were able to create expertise profiles that included data about what APIs a developer may be expert in through their use of those APIs.

2.1.2 Using Interaction to Support Developers

Earlier work from our research group introduced degree-of-interest (DOI) values to represent which program elements a developer has interacted with significantly [52]. The more frequently and recently a developer has interacted with a particular program element, the higher the DOI value; as a developer moves to work on other program elements, the DOI value of the initial element decays. Initial applications of this concept in our research group computed DOIs across all of a developer's workday [51]. Subsequent work scopes the DOI computation per task [52]. In our work on the degree-of-knowledge model, we return to the computation of DOI across all of a developer's work to capture a developer's familiarity in the source across tasks.

Others have considered the use of interaction data for suggesting where to navigate next in the code [22], for tracking the influence of copied and pasted code [66], for informing developers of relevant design defects [65] and for understanding the differences between novice and expert programmers [85]. None of these previous efforts have considered the use of interaction data for determining expertise in or familiarity with source code.

2.1.3 Explicitly Modeling a Developer’s Knowledge

A significant amount of research in the field of psychology focuses on knowledge. Much of this work attempts to create models that describe the different parts of knowledge, such as implicit and explicit knowledge (e.g., [13]). Ultimately, it would be desirable to understand from the neurons up how people remember and learn so that we could use those models to improve how we present information to programmers. However, these neurological foundations are currently too preliminary and too low level to provide much help in this regard. With our work on the DOK model, we aim to determine if we can use one kind of activity—programming—as a proxy for a developer’s knowledge of code without trying to precisely understand how that knowledge is gained and represented in humans.

In the domain of programming, studies have been conducted to explore how programmers comprehend programs, resulting in the proposal of a number of different cognitive models: top-down, bottom-up and integrated. Top-down theories describe the comprehension process as a successive refinement of hypotheses starting from the problem domain and going down to the specific executing problem (e.g., [9, 10]). Bottom-up theories suggest that developers first understand programs in terms of low-level abstractions and then develop higher-level models of it (e.g., [68]). Integrated models combine top-down and bottom-up theories to describe the program comprehension process (e.g., [82]). More recent work by Detienne [23] provides an overview of these approaches and discusses how the particular task, such as reading or manipulating code, effects the understanding. Our focus in the DOK model is different, considering not the mechanism by which programmers learn about a program, but rather what they retain as knowledge as a result of the learning process.

Altmann [3] studied the near-term memory of expert programmers by monitoring them performing a task for eighty minutes. He analyzed a ten minute interval using a computational simulation and studied what was likely entered into the programmer’s memory on a moment-to-moment basis. His focus was to characterize near-term memory, essentially what the programmer could recall less than an hour later. Our focus in modeling a developer’s knowledge is on longer-term memory as the questions driving our work require an understanding of who knows what over the longer-term about the code.

In a more recent study, Ko and colleagues [54] examined program comprehension of developers based on their interactions with an integrated development environment. They found that a significant amount of time is spent on navigating source code and propose to model program understanding as a process of searching, relating and collecting relevant information. Their model suggests that knowledge is not only built by authoring code but also by navigating through it. Again, their focus is on the mechanism of program understanding, our focus with the DOK model is on the knowledge retained as a result. Ko and colleagues work does provide evidence that interaction, through such actions as searching, can play a role in knowledge.

Other studies consider the type of knowledge experts have compared to novices in program comprehension tasks. As an example, Soloway and Ehrlich [77] found that experts have two types of programming knowledge; programming plans and rules of programming discourse. Finally, studies investigated how experts share such knowledge with apprentices and novices (e.g., [5]). These studies do not consider how to determine what knowledge a programmer might have about the source, which is the focus of our DOK model.

2.2 Modeling a Developer’s Information Needs Across Multiple Kinds of Information

With the work on the information fragments model, we are trying to match the individual information needs of a developer across multiple kinds of information. Similar to our empirical study on a developer’s questions (Study_{EXP_FR}), a number of previous studies investigated the questions that developers ask. To answer

developers' questions over complex information spaces, previous work has considered a variety of approaches ranging from navigation to the semantic web. In this section, we first compare our work on developer's questions to previous empirical studies (Section 2.2.1). We then review related work on supporting developer's information needs across multiple kinds of information (Section 2.2.2).

2.2.1 Studies of Developers' Questions

Earlier studies on investigating a developer's information needs have considered questions that developers ask during a software development project. Letovsky gathered think aloud protocols from programmers and identified five general question types within the data: why, how, what, whether and discrepancy questions [58]. Herbsleb and Kuwana focused on studying questions during software design meetings and classified them by the attribute the question referred to (who, what, when, why, and how) [42]. Erdös and Sneed identify seven basic questions from their personal experiences maintaining programs [27]. Johnson and Erdem examined messages posted to the Usenet newsgroup to analyze what information people ask for. They classify the questions as either goal oriented (to achieve task-specific goals), symptom oriented (to identify the source of a problem) or system oriented (to understand the objects and functions of the system) [49]. In a more recent study, Breu and colleagues categorized questions asked in bug reports from the Eclipse and Mozilla projects into eight categories; the categories included missing information and triaging [8]. We contribute to this body of work by identifying the questions that developers have which involve multiple kinds of information and by categorizing the questions in terms of the kinds of information a developer intended to use to answer the questions.

Five recent studies provide more comprehensive question catalogs. Two of these catalogs focus largely on questions about source code. Sillito and colleagues described 44 questions that developers ask when programming [75]. De Alwis and Murphy identified 36 questions from literature, blogs and their own experience that deal largely with source code [20].

The other three studies discuss questions about a broader set of software development activities. Ko and colleagues report on 21 types of questions determined by

shadowing professional developers at work [55]. Some of these question types, for instance, “What have my coworkers been doing?”, focus on problems and issues developers are facing not just in the source code. LaToza and colleagues propose 19 problems from their own experience as software developers and present the results of a survey of professional developers about the seriousness of each problem [57]. In another study, LaToza and Myers surveyed developers and report on 94 hard-to-answer questions on code that developers described [56]. These questions can be classified as either referring to changes, to properties of elements or to relationships between elements. They report that ten of the 94 questions overlap with questions identified in our interview-based study (Study_{EXP_FR}) [29]. While there is overlap in the questions found across these studies, none of the previous studies discusses the ambiguity that lies in the interpretation of questions such as “What have my coworkers been doing?”; a contribution of our work on determining a developer’s information needs across multiple kinds of information is to highlight the personal differences in questions between developers.

2.2.2 Answering a Developer’s Questions

The information fragments model supports a developer in answering questions that span across multiple kinds of information. Previous work on supporting a developer with this type of questions, has explored a variety of approaches ranging from navigation through information to using a solution based on the semantic web. We review these approaches in the following with respect to the information fragments model.

Navigating Through Information

One of the ways that has been proposed to help address questions in complex information spaces is support for navigating through a space by following information links. Feldspar allows a user to follow association links between different kinds of information, such as emails and files, to find information of interest [16]. JQuery provides support specific for software development, allowing a developer to step through structural links between source code elements within a single view, one at a time [46]. These existing approaches are restricted on the navigation paths

that can be followed whereas our approach provides the user with more flexible integrations, and thus navigation, across information.

Querying Information

Other approaches to help answer the questions of interest focus on helping a user express, rather than follow, links between information. These approaches involve a query language. They require a user to explicitly specify the links that exist between the information. Some, such as the relational views by Linton [59] and the approach by Paul and Prakash [67], are based on a relational algebra. Others, such as CodeQuest [39] are based on a database accessed via logic. Yet other approaches support natural language queries: Würsch and colleagues [83] provide such support over source while Devanbu and colleagues [24] support queries over architectural, conceptual and code information. Not all approaches have focused on user-level queries: Garlan considered how to enable views of data in a programming environment to ease the creation of tools to show the information a programmer may desire [34]. All of these approaches require a specific expression by the user or programmer as to how information of interest links together. Different to these approaches, the information fragments model automatically composes information of interest.

Linking Different Kinds of Information

Alternatively, some systems have automatically linked different kinds of software development project information. Hipikat uses a fixed schema to mine information from software project repositories and provides a developer multiple entry points into the created web of information [81]. The STeP-IN system extends Hipikat by including information on programmers [84]. Deep Intellisense automatically displays information relevant to a selected source code element such as change events and the people involved[43]. To provide team awareness, Palantir [71] links change information to source code, by presenting the changes of others in the context of a developer's workspace. FASTDash [7] extends the approach by also visualizing which files are currently open or being debugged by a developer. Like FASTDash, WIPDash [45] also provides a dashboard but focuses on visualizing information

on work items and the team members related to them. All of these approaches are oriented at accessing one presentation of integrated information. Hipikat, for instance, can recommend artifacts related to a provided starting artifact. In contrast, our information fragments model approach focuses on allowing the user to compose and view different kinds of project information to support the many questions that arise during a work day.

Overlaying Information

Several approaches have tried to reduce the necessity of piecing together the information by overlaying and filtering information. Tools such as Seesoft [26] and Augur [33], use a line-oriented visualization of the source code and overlay it with change history and structural information. Other approaches use existing views to overlay information. For instance the original Jazz research prototype [17] annotates the package explorer view of Eclipse with team awareness information. Code Canvas [21] allows the user to add or hide various layers of information, such as code test coverage, execution traces or search results on top of a layer of code documents that is also zoomable. However, all of these approaches take only one aspect of information as the base and overlay other aspects on top of it, without allowing the user to integrate the information the way he wants to.

Composing Information

Ferret by de Alwis and Murphy [20] supports the composition of different perspectives—called spheres in their approach—on similar information. For example, a sphere representing source code can be composed with a sphere representing dynamic information about the system to help a developer investigate such properties as which calls between methods are actually occurring in a run of the system. Their approach focuses on matching similar elements between spheres whereas our approach aims to support the composition of different kinds of information for which links can be automatically determined. Our information fragments model also supports the user in varying the presentation of composed information to enable a suitable interpretation for the task at hand.

Mapping Information in an Ontology

OASIS by Jin and Cordy [47, 48] allows for the integration of different tools. At its core, the approach has a domain ontology that defines the common “conceptual” space for all tools that participate in the integration. Antoniol and colleagues [4] defined an ontology based on bug reports, versioning repository information and source code together with the exact dependencies between the different kinds of information. The specified ontologies are based on or require a fixed one to one mapping between different pieces of information.

Mi and Scacchi use meta-models, similar to ontologies, to define software development models on five types of resources and their relations: software systems, processes, products, tools and agents and discuss the composition of such models. In this approach, the definition of the software development models and the mapping between the resources has to be done manually [61].

In contrast to these approaches, our information fragments model allows for a flexible integration of information, with links between different information items determined automatically based on properties of the information.

Using the Semantic Web

The semantic web attempts to bring structure and meaning to information to allow machines to process it directly [2, 6, 74]. It is based on the Resource Description Framework (RDF) that provides a triple-based language and the Web Ontology Language (OWL) to add meaning to the data. Our information fragment model is very similar in that each node has a unique identifier and properties with certain values, just like the RDF triples and the different kinds of information that a node or a property represent in our model could be described in an ontology. However, instead of using the SPARQL query language or iSPARQL [53]—an extended version of SPARQL for software entities—to query RDF data in the semantic web, our information fragments model approach provides automatic composition and an adaptable presentation of the information.

Haystack is an example of how semantic web technology can be used to support the integration of different kinds of information. Haystack encodes such information as e-mail, calendars and the world wide web using the Resource Description

Framework (RDF) format. Based on a provided user interface ontology, a user can then specify views from different kinds of information. This approach still requires the user to manually specify which information should be presented and how it should be integrated [69], whereas our approach provides automatic composition for many different kinds of information about a software system development.

Chapter 3

The Degree-of-Knowledge Model

In this chapter, we discuss the development of the degree-of-knowledge (DOK) model. The DOK model is a developer-centric model capturing a developer’s knowledge of code that can help to answer such questions as who a developer should ask about parts of the code or who on a team needs to know about changes made. The degree-of-knowledge (DOK) model computes automatically, for each source code element in a code base, a real value that represents a developer’s knowledge of that element.

We begin by presenting the results of our exploratory study, *Study_{EXP_DOK}*, that identify a number of factors that may be used to model a developer’s knowledge (Section 3.1). Next, we present evidence gathered in another study, *Study_{DATA_DOK}*, that shows that both authorship and interaction information about how a developer interacts with the code are important in characterizing a developer’s knowledge of code (Section 3.2). Using the results of *Study_{EXP_DOK}* and *Study_{DATA_DOK}*, we developed the degree-of-knowledge model to capture a developer’s individual knowledge of code based on a developer’s authorship and interaction information. We describe the DOK model in Section 3.3. Finally, we conducted an experiment, *Experiment_{DOK}*, to determine the relative effect of authorship and interaction towards modeling a developer’s knowledge. We present the results in Section 3.4.

3.1 A Developer’s Activity as an Indicator for Knowledge

As an initial step in understanding what factors contribute to a developer’s knowledge of code, we conducted an exploratory study, which we refer to as Study_{EXP_DOK}. In this study, we were particularly interested in whether a developer’s interaction can be used as an indicator for his knowledge of the structure of the source code. After all, the idea that a developer gains knowledge of code by reading or editing the code is consistent with cognitive models on program comprehension [23]. To determine whether factors other than interaction contribute to a developer’s knowledge of code, we interviewed the participants at the end of the study. A more detailed description of Study_{EXP_DOK} is presented elsewhere [31].

3.1.1 Study

The hypothesis for Study_{EXP_DOK} was that *the more frequently and recently a developer has interacted with a particular source code element, the higher the developer’s knowledge of that element*. To conduct this study, we needed a means of asking a developer about their knowledge of code. We chose to use the structural relations of a code element as a basis for our inquiry because we believe that this kind of knowledge plays an important role in being able to explain how the code works, and also because the structural relationships of an element are automatically determinable, allowing the study of a larger set of developers.

To evaluate our hypothesis, we monitored the interactions of nineteen professional Java developers with their integrated development environment (IDE) over a period of five weeks. When a developer reached a certain threshold of interaction with the environment, an automatically generated questionnaire appeared that asked the developer about the structure of code elements with which the developer had interacted. There were three thresholds, and thus three questionnaires, that a developer might be asked to answer. To ensure our study did not stress memory recall, some tool support was provided with each questionnaire to help a developer answer a question about a code element. At the end of the study period, we interviewed developers to determine factors that may contribute to model of a developer’s knowledge of code. Thirteen of the nineteen developers volunteered to be interviewed.

Subjects

Our study involved professional Java developers recruited from two sites of the same company. To be eligible to participate in the study, a developer had to use the Eclipse IDE¹ in his daily work.² To solicit participation, we advertised our study at both sites in a short presentation and randomly asked people at the two sites. We ended up with nineteen subjects that had the appropriate tool environment and sufficient time. Eight of the nineteen subjects were at one development site and eleven were located at the other development site. Members of the former group worked on two different development teams and projects. Members of the latter group were spread evenly across six different development teams. The experience of these subjects ranged from one month to twenty years of professional software development ($M = 7$ years, $SD = 6.4$ years)³. Two of the nineteen subjects were female.

Method

Our overall method involved monitoring the interaction of subjects with the Eclipse IDE and prompting the subject with a questionnaire about pieces of the system structure with which he had interacted when certain thresholds of interactions were reached. The interactions we monitored included selections and edits of source code, and commands, such as the opening and closing of editors, views and perspectives (see Table 3.1).

Interaction Monitoring. Our goal was to have each subject answer approximately one questionnaire per week over a three week period. To ensure that the number of interactions was approximately equal between each questionnaire, we based the questionnaire prompting on the number of interaction events. We assumed an average interaction event number of 4000 per day.⁴ To approximate one questionnaire

¹www.eclipse.org/, verified 18/11/10.

²While we use the pronoun *he* in describing the study, the study involved both male and female participants.

³ M , SD stand for mean and standard deviation respectively.

⁴We made this assumption based on the average interaction events of a professional software developer who we monitored for two weeks while working on an open source tool; we reduced this developer's interaction events by one-third to account for more interruption time in a co-located

Table 3.1: Monitored Interaction Events (see [52] for more detail)

Event	Description
<i>selection</i>	Selections in an editor or view of the IDE via mouse or keyboard.
<i>edit</i>	Textual and graphical edits in an editor of the IDE.
<i>command</i>	Operations such as opening or closing of editors, saving, building or setting preferences.

per week, we thus chose 20000, 40000 and 60000 interactions as thresholds. Once the threshold was exceeded, a dialog in Eclipse opened automatically and the subjects could either work on the questionnaire or postpone it. A questionnaire was always generated with the most recent 20000 (40000 or 60000 events), therefore, postponing a questionnaire did not affect the recency of exposure to the material asked about in the questionnaire.

Questionnaire Content. Each questionnaire presented to a subject contained eighteen questions about the source code elements with which the subject had interacted. Each questionnaire was generated specifically for a subject based on the interaction captured for that subject. The eighteen questions were divided into three categories of six questions. Each group of six questions contained a question about type hierarchy (Q1 in Figure 3.1), two questions about type parameters (Q2), and three questions about inter- and intra-class relations (Q3 and Q4). We chose these detailed structural questions as a starting point for investigating knowledge because we could determine the correctness of answers to these questions objectively. A questionnaire asked these six questions for three different sets of elements: one set with high DOI, one set with medium DOI and one set with low DOI values. To avoid learning effects, the method and type elements were chosen so that no element was asked about twice (over one questionnaire and over all questionnaires).

group environment.

- Q1 Can you recall the name of one class or interface that is directly extended, implemented by ‘*TYPE*’, or can you recall the name of one class that directly extends the type ‘*TYPE*’?

Q2 Do you know the types of the parameters that are passed to the invocation of method/constructor ‘*METHOD*’?

Q3 Do you know two methods that are called by method/constructor ‘*METHOD*’?

Q4 Can you recall one method/constructor that calls method/constructor ‘*METHOD*’?

Figure 3.1: Types of Questions in the Questionnaire

Degree of Interest. We use a real number value to represent the amount of activity (selections and edits) a developer has had recently with a particular program element. This value represents the degree of interest (DOI) of the element [50, 52]; it is a combination of two components, a frequency of how many interactions a developer has had with the element and a recency that decays the DOI value based on the number of interactions a developer has had with other source since the last interaction with the element of interest. The DOI of an element starts at a positive value with the first interaction. If a developer continues to interact with that element, its DOI will rise. If a developer ceases to interact with the element, its DOI will gradually decay until a developer again begins to interact with it. Different kinds of events contribute different scaled values to the DOI of an element; for instance, selections of an element contribute less to DOI than edits of an element. The DOI function used in this study has been used successfully as part of the Eclipse Mylyn⁵ project, which is supporting hundreds of thousands of Java programmers in their daily work. In contrast to Eclipse Mylyn, our use of DOI considers all interaction a developer has with the environment and does not consider any task boundaries indicated by the developer as part of their work.

⁵www.eclipse.org/mylyn, verified 19/11/10

Study Support

We implemented the support for the study as an Eclipse feature⁶ based on the Mylyn monitor [64], a Mylyn component for computing DOI values [52], and the questionnaire component. When a threshold is exceeded, the questionnaire component determines elements (method and type elements) with high, medium and low DOI values and generates the questionnaire. The component considered the 20% of elements with the highest DOI, the 20% in the middle and the 20% of elements with the lowest DOI value as the group of elements with high, medium and low DOI values. The component takes into account elements that were touched (i.e., selected or edited) at least three times to avoid elements touched by accident.

To form the questions, elements were randomly drawn from the groups of high, medium and low DOI. We placed constraints on the elements used for a question to make sure answers were possible. For example, for Q3 each of the method elements selected had to have at least two method calls in their method body. Once eighteen suitable elements were determined, the questionnaire component opened a wizard dialog that had one page for each question. To ensure our study did not stress memory recall and facilitate the answering process, some tool support—the open type action and the outline view of Eclipse—was provided with each questionnaire to help a subject answer a question about a program element (Figure 3.2). The answer fields of the question pages could be filled by selecting an element in either the open type view or the outline view.

Result Scoring

Twice per week, we visited each subject and collected the relevant data and manually analyzed the correctness of the subject's answers with all possible answers found by our tool support. To ensure our tool also found all possible answers, we compared our results to the actual code bases on a subject's computer for a random sample of subjects. We then summed up the number of correct answers for each group of six questions (low, medium, high). In each group of six questions, Q1 and Q4 were asked once with one possible answer each. Q2 was asked twice, once about a method with two parameter types and once with one parameter type, result-

⁶The subjects used a variety of versions of Eclipse from 3.2.1 to 3.3M4.



Figure 3.2: Questionnaire with Open Type Action

ing in two possible answers in the first case and one possible answer in the second case. Q3 was asked twice with two possible answers each time. Therefore, the correct answer score for each group was between 0 and 9. For each questionnaire, we thus computed three values ranging from 0 to 9, one for the group of questions about elements with low DOI, one for medium and one for high DOI.

Interaction Levels

There was a substantial difference in the period of time it took each developer to reach the first threshold. Several developers reached the first threshold after three days, the second after six and the third after around nine days. Other subjects took twelve days of programming to reach the first threshold. The mean time for

reaching the first threshold was 7 ($SD = 3.10$).

Operational Problems

We also faced several operational problems when doing our study. Eight subjects that had completed the first questionnaire did not reach the second questionnaire and only eight subjects reached the final questionnaire. This attrition seems to be for one of three reasons. First, some subjects accidentally uninstalled our plug-in by installing a new version of Eclipse and deleting the old workspace in which the data was stored. Second, there was insufficient time from when a subject joined the study to the end of the study period to collect the results. Third, a questionnaire could not be created. For example, two subjects reached the second interaction threshold without having completed the first questionnaire as the study plug-in was unable to create the first questionnaire due to interaction with an insufficient number of elements to fulfill the criteria for creating the first questionnaire.

3.1.2 Quantitative Results

To evaluate our main hypothesis of whether the frequency and recency of a developer's interaction with particular parts of the source indicates knowledge of that source, we conducted within subject statistical tests. Per subject, we paired the number of correct answers for the six elements with a low DOI within a questionnaire with the number of correct answers for the six elements with a high DOI within the same questionnaire. We did not consider the elements with medium DOI in the questionnaires because the range of DOI values for those elements was very close to the range for high and low DOI elements. In our presentation of the data, we have altered genders so as not to identify the subjects.

We depict the results of correct answers for the elements of low and high DOI for each subject for each of the questionnaires in Figure 3.3. We have arranged the results per subject so that the difference between the correct answers for high DOI and low DOI elements increases from left to right; the differences are shown in the trend line overlying the histogram. For the subjects where the trend line is above zero, the data supports our hypothesis that a programmer's activity, modeled by DOI, indicates knowledge.

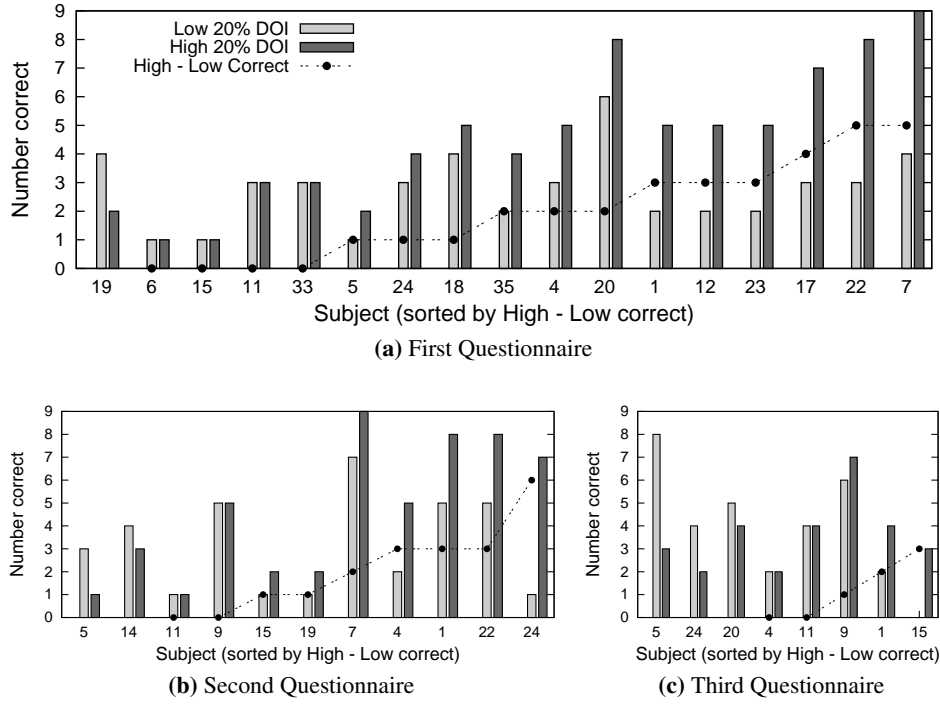


Figure 3.3: Correct Answers by Subject for Questionnaire 1, 2 and 3

To determine whether there was statistical significance in the mean difference between correct answers for high and low DOI elements, we performed paired t-tests by pairing the number of correct answers for high and low DOI elements by subject. The difference in the number of correct answers for each questionnaire passed the Kolmogorov-Smirnov and the Shapiro-Wilk normality tests.

First Questionnaire

Seventeen subjects answered the first questionnaire.⁷ Using a paired t-test, we found that the mean high DOI correct is significantly⁸ higher than the mean low DOI correct per subject (two-tailed $p = 0.0016$, $DF = 16$ ⁹, mean difference = 1.7647).

⁷ Although the interaction of two more subjects caused the subjects to pass the first threshold, the study plug-in did not find enough elements in their interactions to create the questionnaire.

⁸ We consider results to be statistically significant with $p < 0.05$.

⁹ The degrees of freedom (DF) represent the number of observations used to estimate a parameter (for example, the differences between mean low and high DOI correct per subject). In this case,

Second Questionnaire

Eleven subjects reached the threshold for the second questionnaire. For the data provided by these subjects, the two-tailed P value for the paired t-test is 0.0578 and is thus not significant at the 95% significance level ($DF = 10$). We hypothesize that this lack of significance is from the lack of data: there are only eleven observations in the second questionnaire versus seventeen from the first.

Third Questionnaire

Only eight subjects reached the third threshold. Performing the paired t-test on the results of this questionnaire did not deliver a significant result. For this questionnaire, we saw more evidence of individual factors influencing the results. Subject S24 stated that the third questionnaire was the hardest because there were hardly any questions about elements that were written by her. In fact, most elements with a high DOI for that questionnaire seemed to gain a high DOI because she stepped through them a lot when debugging. Subject S4 and S9 stated that all six high DOI elements asked about in the questionnaire were not written by them and they thought it was interesting and even “surprising” (S9) that those elements had a high DOI. Similarly to S24, subjects S4 and S9 stated that they had interacted with several of the high DOI elements as a step of the debugging process. Furthermore, these two subjects said they should have known better the six elements with low DOI because they wrote or edited those elements significantly.

Summary of Questionnaire Results

Assuming the answers from a given subject are independent across questionnaires, we tested whether the mean high DOI correct for a subject is significantly different from the mean low DOI correct. Using a paired t-test, we found that the mean high DOI correct is significantly higher than the mean low DOI correct per subject ($p = 0.0024$, $DF = 35$, mean difference = 1.22). Although on average a subject only has about one more correct answer for high DOI questions than low DOI questions, this result supports our hypothesis that a programmer is more likely to have knowledge of program elements with which he or she has frequently and

DF = number of subjects - 1.

recently interacted.

Further Results

The DOI value is an aggregation of several primary components: selects,¹⁰ edits and recency. To investigate their impact, we plotted the correct answers with respect to these components and performed regression analysis. However, there appears to be no trend in correct and incorrect answers for the components of DOI even when breaking the data down by subject.

Over all subjects, the mean of the DOI values for the elements in each group (low, medium, high) were in a close range. This result holds for the data from all three interaction intervals. Thus, even though subjects differ in the systems on which they are working and their tasks, the range of DOI values is similar. No matter which subject we consider, if we take the 20% of elements with highest, medium or lowest DOI for a certain interaction interval (e.g., 20000 events), the DOI values for the set of elements are in a similar range.

The number of interactions per day per subject ranged from 1850 to 8550. A reasonable assumption might be that the more a subject programs in a day, the better the subject would know the code. However, we do not see any significant impact on the results despite this large difference.

3.1.3 Qualitative Results

We were able to interview in-person thirteen of the nineteen subjects after each had completed the questionnaire portion of the study.¹¹ These are the subjects who volunteered to be interviewed. Each interview took between five and thirty minutes depending upon the time available from the subject. We also received some data in response to email from subjects S1, S6 and S7. Each interview started with some general questions (see Table 3.2), followed by a variety of questions based on a subject's response to previous questions.

¹⁰A *select* event is generated when a developer selects (clicks on) a code element to apply a command, such as navigating from a use to a definition.

¹¹Interviews were conducted with S4, S5, S9, S11, S14, S15, S17, S18, S19, S20, S22, S24, and S33.

Table 3.2: General Interview Questions

1	Do you think that the correctness of your answers reflects what you know about those elements?
2	Do you think there are some elements for which you should have known the correct answer and/or are there some elements where you are surprised that you knew the correct answer?
3	Why do you think you knew more about the elements that were answered correctly than about the other elements?
4	What kind of knowledge do you think you keep most in your mind over a long and/or short period of time?

Subjects' View of the Results

We wanted to know whether the results of our study reflect what a subject believes he or she knows about the elements asked about in the questionnaires. To investigate this issue, we showed each subject a list of the program elements that appeared in all of the questionnaires that the subject had completed; this list indicated for which of the elements the subject answered the question(s) correctly. We then asked the subject if the split between elements for which he or she answered questions correctly and those answered incorrectly reflects what he or she knows about the elements. We then went over each element, asking the subject whether the correctness of the answer surprised him or her.

Overall, eleven of the sixteen subjects (69%) whom we interviewed stated that the correct versus incorrect answers represented “fairly well” what each subject knew about the code and in one case, was “exactly” what the subject knew (S20). Two subjects (12%) stated that the first and second questionnaire was fairly reflective but the third questionnaire was not. Three subjects (S5, S19, S33) (19%) stated that she or he should have known the correct answer to some elements because she or he wrote the code. These three subjects also expressed surprise about having a correct answer to several elements.

Subjects' View on Their Knowledge

In the interviews, we also asked what kind of knowledge a subject thought she had about the program elements and what the subject thought about the questions. Most

subjects (S1, S4, S7, S14, S18, S19, S22, S24, S33) (56%) responded that he or she would know what a particular method asked about does, but that the questions in the questionnaires were too detailed. Three subjects (S14, S18, S33) stated he or she would know the flow of control in the program, but he or she would not necessarily know the direct calls, only that there was some collaboration between two elements. As one subject stated, “remembering finer details isn’t my strong point” (S19).

Authorship and Editing

All subjects stated that he or she knows more about elements he or she authored. When we asked the subjects about the questions they answered correctly, they mostly stated those elements were the ones they had authored. As one subject [20] explained,

when you write your own code you follow your own patterns so it is easier to know afterwards, [...], you can tell how you would have done it.

Two of the subjects each further stated that most of their correct answers occurred for code they wrote recently (S1, S14). One subject (S15) said that authoring the (Java) classes would probably cause one to know those classes better for one to two months. Another subject (S1) stated the opposite, saying,

I would say that global knowledge of the system is maintained over a longer period of time but the specifics of each method implementation deteriorates quite quickly [...] if I was asked the same questions now [2.5 weeks afterwards], I would get most of them wrong.

Overall, there was a large discrepancy about the period of time that subjects thought they would know about code that he or she authored from three months (S22) to one and a half to two years ago (S5, S11). These latter two subjects each noted that his knowledge was dependent on how long it took him to write the code—the longer the authoring time the better the knowledge—and whether he was actively maintaining the code. Several subjects noted that he or she has to “work with code [continuously] otherwise I forget after a while [1 month]” (S18).

Code Stability

The stability of the code also has an influence on how long a subject knows about the program elements. Both subjects S11 and S5 were authoring code at a low level of the system; this code needed to be robust and does not change often. These two subjects each felt they had good knowledge of the code one and a half years after the code's creation. Other subjects, who were working on code that was changed more frequently, stated that they would not know their code more than a couple of months.

Role of a Program Element

Several subjects (S1, S7, S9, S11, S15, S19, S22) (43%) each mentioned that he knew more about elements that played a more important role in the code. For instance, if a class was a hub of an API, the subjects (S1, S11, S22) would know the correct answers to questions posed, whereas internal code was less known. Subjects also stated that they would know the abstract classes on the top of the type hierarchy or the root super class in general but they did not know about intermediate classes (S9, S19). Furthermore, if a class was part of a test, it seemed not to be as important and therefore not to be known as well as other code (S7).

Task Locality

When we interviewed subjects about the elements, they would often refer to a set of elements as a task or talk about a general task. One subject even remembered the overall tasks three weeks after the actual questionnaire (S11). Some subjects identified the kind of tasks undertaken over the period of the questionnaire as a reason for not having a lot of correct answers. From one (S11), "I was into code all over this place trying to thread through some of the stuff I was working on". Another (S33) explained that he fixed a lot of small bugs and explored a lot of code very briefly that he has not written and therefore he did not know the answers. Yet another (S9) explained that he worked on crosscutting changes, spread over several modules, in other people's code. When working on other people's code, subjects described that they were focusing more on getting it to work than understanding how it works (S14, S15, S19). One subject (S1) also noted that if the task was to

create a new feature, she knew the code more than if it was a refactoring task.

Short-term Activity

Similarly to the overall task, the activity undertaken to complete a task in the short-term with just a small subset of elements influenced knowledge of the elements. For instance, one subject (S11) said,

It depends upon what you are doing, if you are fixing a bug you are concentrating directly in there but if you are just adding an extra parameter to pass it through to something deeper you don't know a whole lot about what's going on in there.

For nine subjects (S4, S9, S14, S15, S19, S20, S22, S24, S33) (56%), debugging activity heavily influenced the subjects' responses to the questionnaires because elements appeared in the questionnaire as a result of activity stepping through the elements repeatedly during debugging. This activity was intense but did not consider the structure or functionality of the element. A similar situation occurred when going through lists of search results.

IDE

Integrated development environments, such as Eclipse, provide substantial support to find structural information about code. Four subjects (S6, S14, S15, S22) each stated that he relies heavily on these structural determination tools, "I live by the call hierarchy view" (S22). These subjects also noted that the presence of these tools likely causes them to remember less about the kinds of information that can be retrieved fairly easily and quickly (S15),

in a VI text editor I would probably know more about the actual calls but in Eclipse I have the JDT support to help me.

Although our questionnaires provided access to some tools, for finding types and methods, the lack of access to all tools created a "disconnect feeling" (S14) to the code.

Code Patterns

Patterns in code facilitate knowledge for some subjects (S11, S22). As one (S22) stated, “I know what this method does because it always does the same for each class it is in”. When showing one correct element to one subject (S11), she was surprised to have known the answer, but once she thought about it she said, “I did not know this well [but] that’s our pattern for writing some of the tests”. This situation also arose for another subject (S7) who stated that “a lot of our method signatures contain common data structures which are easy to recall [patterns]” (S7). These common patterns are similar to clichés [70].

3.1.4 Threats to Validity

The validity of our results is threatened by several factors. A primary threat to the validity of our study is the number of subjects. In particular, for the second and third questionnaire, the number of subjects that completed the questionnaire is small, possibly skewing the results as we begin to look at a much smaller set of programmers.

The validity of our results is also threatened by undertaking the study in situ rather than in a laboratory. In situ, there are many variables which we can neither control nor account for, such as the type of work being performed by the subject. This lack of control shows up in several ways in our results. For example, for subject S24, the number of correct answers for low and high changed substantially over the three questionnaires (Figure 3.3) because of the nature of her activity during the different periods. As another example, subject S5, who had substantially more correct answers for the low DOI elements in the second and third questionnaire than for the high DOI elements, stated that for these questionnaires, there were more elements that were written by him in the low DOI elements than in the high DOI elements. Since he stated he believed he knew everything he wrote, this provides a possible explanation for his results for these questionnaires.

The validity of our results are also threatened by measuring activity with the source only through the programmer’s interaction with the development environment. We chose to monitor the IDE because it is a common tool used by all programmers. Other forms of activity, such as design meetings, are much more diffi-

cult to monitor and to conceive of using as a basis for subsequent tools. Although our monitoring of the IDE was extensive, it was aimed at certain mechanisms that provide good coverage of textual editing, selections, and so on. Our monitoring misses interactions through graphical editors which some of the subjects may have been using. In these cases, the DOI values we assigned would not be representative of the actual interaction and activity of the subject with the source. The interviews we conducted with the subjects did not highlight any of these issues as seriously compromising the study.

3.1.5 The Need for a Better Indicator of a Developer's Knowledge

The results of our study show that a developer's interaction with the code can help indicate the developer's knowledge about source code. They also suggest that additional factors should be used to augment DOI to gain a better indicator for a developer's knowledge of code. As most developers stated, authorship of code is a significant factor in their knowledge of the code, supporting the tacit assumption inherent in current expertise recommenders (e.g., , [63]) that are based solely on authorship. This suggests that a knowledge model should take into account a combination of both interaction and authorship. Furthermore, the developers' comments on the ebb and flow of knowledge suggest, rather than taking a global expertise approach, a model that takes an individual perspective on a developer's knowledge. Such an individual model can then capture the ebb and flow for each developer that occurs in particular when code is changed frequently (code stability). Other factors mentioned by developers in the interviews, such as the role of a program element or the short-term activity, are more difficult to determine automatically and thus difficult to integrate into an indicator that can be determined automatically. In the following, we will investigate whether interaction and authorship indeed capture different aspects and could thus lead to a better indicator of a developer's knowledge.

3.2 Authorship and Interaction

To investigate the possible effects of authorship and interaction on a developer's knowledge, we gathered data in a new study, Study_{DATA_DOK}, from the profes-

sional development site we refer to as Site₁. In particular, we were interested in the following three questions:

1. how do authorship and interaction vary across developers,
2. do any patterns emerge about a developer's authorship and interaction, and
3. do authorship and interaction capture two different aspects of a developer's work.

We answer the first two questions by presenting the data (Section 3.2.1 and 3.2.2). To answer the third question, we must interpret the differences in the authorship and interaction data (Section 3.2.3). In our presentation of the data, we use the median and the range to present data that summarizes data over all developers where there is a big deviation between developers. We use the mean and the standard deviation for data presented per developer.

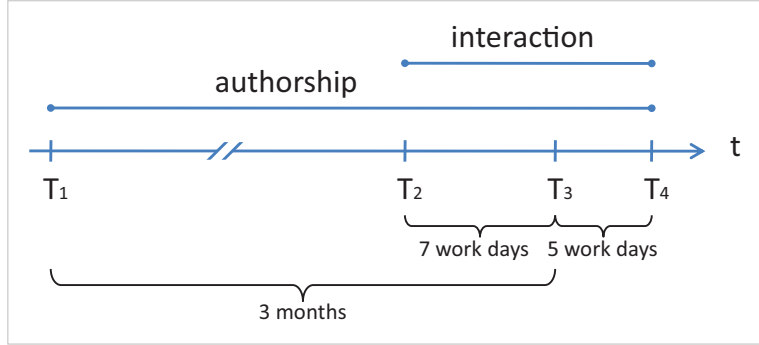
Subjects. Site₁ involved seven professional developers (D1 through D7)¹² building a Java client/server system, using IBM's Rational Team Concert (RTC)¹³ system as the source repository. The professional experience of these developers ranged from one to twenty-two years, with a mean experience of 11.6 years ($SD = 5.9$ years)¹⁴. These developers each worked on multiple streams (branches) of the code; we chose to focus our data collection on a developer's major stream. One developer (D5) could not identify a major stream of the four on which he worked; as this work pattern makes authorship difficult to determine, we have chosen to exclude his data from the presentation given in this section but have included his results in our experiment (Section 3.4) and case studies (Section 4.1). We discuss the issue of multiple streams, which threatens the validity of our study, further in Section 4.3.

Timeline. Figure 3.4(a) provides an overview of the different periods of data collection. Authorship information was gathered for a three month period (T_1 to T_3).

¹²Five of the seven developers also participated in our initial study presented in Section 3.1.

¹³IBM's Rational Team Concert is a team collaboration platform built on top of Eclipse; the subjects used version 2.0 of RTC on top of Eclipse 3.4.; jazz.net verified 19/11/10.

¹⁴ SD stands for standard deviation



(a) An Abstract Timeline

	T_1	T_2	T_3	T_4
Site ₁	3/11/2008	22/1/2009	2/2/2009	7/2/2009

(b) Specific Points in Time Used

Figure 3.4: Data Collection Time Periods

The interaction data used for our experiment that we report on in Section 3.4 was gathered over seven working days (T_2 to T_3). Case studies that we report on in Section 4.1 were conducted during the period T_3 to T_4 . The data reported on in this section is from data collected from T_1 to T_4 . Figure 3.4(b) maps abstract time points to particular dates used for this development team.

3.2.1 Authorship—a Long-term Component

From a developer’s perspective, a day in a collaborative environment involves many different kinds of authorship events. Consider a developer that creates a new method and changes another existing method to accomplish a task. When he flows the changes to the source code repository he shares with his team, the creation of the new method and the change to the existent method each result in an authorship event. When he updates his local workspace at the end of the day, he accepts the changes a team member made to the source code resulting in one authorship event for each code element the team member changed.

Since developers in Study_{EXP_DOK} differentiated between authoring new code and maintaining existing code, we distinguish between three different kinds of au-

Table 3.3: Average of Six Weeks of a Developer’s Authorship Data Over a Three Month Period (FA = First Authored, DL = Delivered, AC = Accepted, Change stands for the change in elements per week)

Subj.	# Events			# Distinct Elements	
	FA	DL	AC	FA & DL	Change
D1	453 (± 105)	285 (± 66)	62463 (± 5320)	567 (± 113)	10% ($\pm 9\%$)
D2	297 (± 128)	154 (± 43)	51183 (± 5457)	417 (± 160)	15% ($\pm 16\%$)
D3	398 (± 159)	445 (± 112)	62358 (± 5467)	678 (± 214)	18% ($\pm 22\%$)
D4	31 (± 34)	9 (± 10)	63730 (± 5934)	36 (± 39)	14% ($\pm 38\%$)
D6	907 (± 898)	481 (± 43)	61914 (± 5677)	1084 (± 874)	14% ($\pm 10\%$)
D7	98 (± 164)	399 (± 725)	75540 (± 11993)	485 (± 868)	42% ($\pm 50\%$)

thorship events with respect to a developer D :

1. first authorship, representing whether D created the first version of the element¹⁵,
2. deliveries, representing subsequent changes after first authorship made to the element by D ,
3. acceptances, representing changes to the element not completed by D .

We found that the authorship of code loaded into a developer’s environment changed frequently. At this site, a first authorship, delivery or accept event to an element occurred on average every 54 seconds.

For a better understanding of a developer’s authorship and its fluctuation, we counted events and elements over a sliding window of six weeks. We started with the first six weeks and then slid the window eight times, each time by one week until we reached T_3 . Table 3.3 reports numbers for each developer averaged over the eight data points.

First Authorship Events. The number of first authorship events developers produced over a period of six weeks ranges from 0 to 3046 with a median of 323 over all data points collected. This high variation between different developers is not

¹⁵In this thesis, the term element stands for a class, a method or a field.

surprising given the different roles of the team members. Some team members have a more coding centric job, others carry out a more managerial role (see Section 4.3.4). This high variation can also be seen in the differences in the means of FA events over developers in Table 3.3. For some developers this variation happens on a weekly basis. These variations are due to the merging of streams (see Section 4.3.2), differences in project phases, such as testing weeks (see Section 4.3.3), or the fact that some developers seem to work for longer periods of time individually before sharing the work with others through the repository. For example, developer D7 did not have any first authorship events for the first five points in time we counted the events. In the end however, the number of first authorships increased to 396, indicating that he waited for several weeks before committing his changes. These individual variations can also be seen in the standard deviation being relatively high compared to the mean in Table 3.3.

Delivery Events. These same variations between developers as well as within developers occur for delivery events. The median of delivery events is 230 with the actual number of delivery events that a developer had over a six week period ranging from 0 to 1601.

Accept Events. In contrast, the number of accept events is relatively stable over all developers, with a median of 62500 and a range from 45136 to 91985. The numbers presented in Table 3.3 show that the standard deviation for accept events is usually less than 10% of the mean value. This stability stems from the balancing effect of the whole team, which is bigger than the seven developers we looked at, committing changes to the same project.

All Authorship Events. Over the three months and the six developers, there is a ratio of 95 to 1 for accept events versus all first authorship and delivery events. This large ratio is indicative of the high rate of change occurring to elements in a developer's environment.

Authored Elements. These aggregate statistics on first authorship, delivery and accept events count multiple events happening to the same element. Considering unique elements, the developers first authored a median of 323 elements ranging from 0 to 3011. This data follows the same trend as the data on first authorship events. The slight difference in the range between first authorship events (median: 323; range: 0 to 3046) and elements first authored is caused by developers merging streams. The number of unique elements that a developer delivered changes to (median: 161; range: 0 to 1551) is 1.4 times smaller than the number of delivery events, since, on average, developers delivered more than one change to the same element. The same occurs for unique elements that developers accepted changes to (median: 33868; range: 22058 to 42809), which indicates that on average a developer accepts two changes to one element every six weeks. Thus, each day, a developer authored eight new elements, delivered changes to five elements, and accepted changes to 792 elements on average over the period T_1 to T_3 .¹⁶

Change in the Set of Authored Elements. To get a sense of the fluctuation in the set of elements a developer first authored and delivered over six weeks, we compared the sets for consecutive weeks. Over the eight data points we gathered for each developer, on average, 19% of the elements in the set changed each week. Developer specific numbers of the change are presented in the last column of Table 3.3.

3.2.2 Interaction—a Short-term Component

To gain a better understanding of a developer’s interactions, we looked at a period of five work days (T_3 to T_4). For each of these five days, we counted the developer’s interactions over the previous seven business days and then averaged the numbers over the five days (T_3 to T_4) to account for fluctuations within a work week. Table 3.4 reports numbers for each developer averaged over the five days.

¹⁶These numbers differ from the ones presented in [32]. To account for daily anomalies, such as days with lots of meetings, we averaged over eight six week periods instead of averaging over five days, each taking into account the past three months.

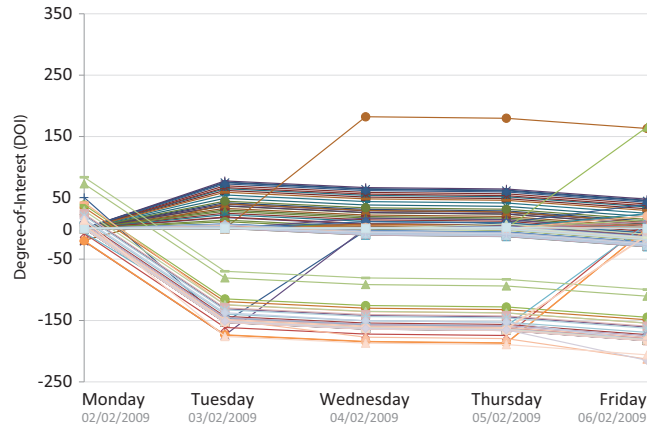
Table 3.4: Developer’s Interaction Data Averaged Over Five Work Days

Subj.	# Interaction Events (over 7 days)	# Distinct Elements (over 7 days)	# Els with pos. DOI (per day)	Change in pos. Els (per day)
D1	11948 (± 570)	593 (± 88)	51 (± 9)	59% ($\pm 28\%$)
D2	4687 (± 1522)	700 (± 112)	36 (± 11)	65% ($\pm 23\%$)
D3	11526 (± 565)	1550 (± 105)	42 (± 1)	81% ($\pm 7\%$)
D4	5700 (± 883)	899 (± 179)	52 (± 14)	75% ($\pm 16\%$)
D6	7470 (± 2863)	974 (± 208)	48 (± 9)	57% ($\pm 41\%$)
D7	8219 (± 1233)	1258 (± 167)	40 (± 17)	73% ($\pm 11\%$)

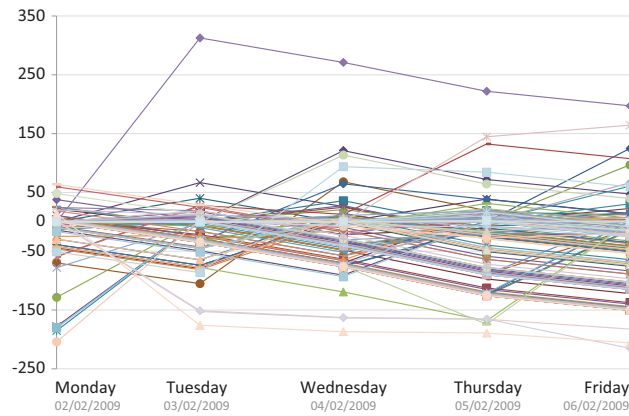
Interaction Events and Elements. We found that developers interacted with many different elements over a week of work, some of them quite frequently. The number of interaction events of a developer over a period of seven business days ranges from 3296 to 12841 with a median of 7823. Considering unique elements, a developer interacted with a median of 898 elements ranging from 514 to 1651. As with the authorship information, the difference between individuals is quite substantial as it depends on the individual’s role on the team and their individual work patterns (see Table 3.4). Different to authorship information, the variation in interaction for a single developer over the five days is relatively small with the standard deviation, on average, only being 15% of the mean for the unique elements.

Degree-of-Interest. One way to indicate a developer’s ongoing interest in a particular code element is to consider a degree-of-interest (DOI) real value for the element computed from the interaction information. We provided an overview of DOI in a previous section (Section 3.1.1) and DOI is reported in earlier work [50, 52]. A positive DOI value suggests that a developer has been recently and frequently interacting with the element; a negative DOI value indicates a developer has been interacting with other elements substantially since the developer interacted with this element.

Change in the Set of Elements with Positive DOI. At this development site, each developer had 45 ($SD = 12$) elements with a positive degree-of-interest per day,



(a) D1



(b) D3

Figure 3.5: Positive DOI Elements

taking into account the sliding window of seven business days. On average, 68% of these elements with a positive DOI changed per day (see Table 3.4). We can see the stability of the size of the set of elements with a positive DOI as well as the high fluctuation of the set's elements in graphs we produced for two developers. Figures 3.5a and 3.5b show, for the period of five working days (T_3 to T_4), elements with a positive DOI value on at least one of the five days for each of the

two developers.¹⁷ For both developers, the size of elements with a positive DOI stays fairly stable over the work week at 51 ($SD = 9$) and 42 ($SD = 1$) for D1, D3 respectively. However, the elements within these sets change substantially per day. For developer D1 59% ($SD = 28$), for D3 81% ($SD = 1\%$) of elements change each day.

These graphs also show the differences in work patterns across the elements for different developers. Some developers, such as D1 (Figure 3.5a), continuously interact with a group of elements, which results in many lines above zero. Other developers, such as D3 (Figure 3.5b) interact with more elements less frequently, resulting in more lines below zero due to the decay of interest in elements. Most of the six developers also had at least one code element with which he or she was interacting with a lot more than with the rest of the code.

3.2.3 Authorship and Interaction

While 68% of the elements with a positive DOI change each single day, only 19% of the elements a developer delivered changes to or first authored change per week. The high change rate for positive DOI elements per day suggests a developer does not interact with specific elements for very long, lowering the likelihood a developer knows the elements. This data is consistent with developers' comments on only knowing code if it was visited shortly before, but knowing it for a long time when they are the first author (see Section 3.1.3), indicating that interaction models a short-term component of a developer's knowledge of code and authorship models a more long-term component of it.

Numbers on distinct elements in Table 3.3 and Table 3.4 show that interaction captures elements not authored. Even though the numbers on distinct elements in the authorship table and the interaction table are computed using different sliding windows, in general, developers interacted with more distinct elements over a seven day period than they authored over a 6 weeks period. This fact is particularly evident for developer D4 who first authored or delivered 36 elements over six weeks, but interacted with 899 distinct elements over 7 business days.

An analysis of a developer's work week also suggests the notion of author-

¹⁷The DOI values shown in these graphs were based on the prior seven days of interaction for each day indicated.

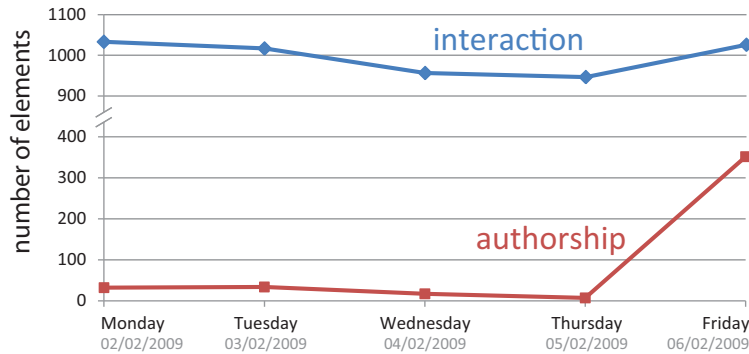


Figure 3.6: Authorship and Interaction over Five Days

ship and interaction capture different aspects of knowledge. The plot in Figure 3.6 shows the number of elements with at least one interaction event over the past seven business days and the number of elements with a delivery or first authorship event over the past three months, for each of five consecutive days (T_3 to T_4).¹⁸ Although the number of elements developers interacted with holds relatively steady, the number of elements delivered or first authored increases prominently on Friday. This data suggests that the developers *created* changes throughout the week but *delivered* most of them on Friday. Interaction may thus be a more useful predictor of recent knowledge whereas authorship helps capture a developer’s knowledge of code over a longer period of time.

Summary. Authorship and interaction capture different sets of elements and each can contribute valuable information to representing a developer’s degree-of-knowledge for a source code element. Interaction models a shorter-term component of a developer’s knowledge of code. Authorship models a longer-term component of knowledge.

¹⁸We used all data available for this analysis: 3 months of authorship and 7 business days of interaction.

3.3 Degree-of-Knowledge Model

Based on our initial study (Study_{EXP_DOK}) and the evidence from the data (Section 3.2), our definition of DOK includes one component indicating a developer’s longer-term knowledge of a source code element, represented by a degree-of-authorship value, and a second component indicating a developer’s shorter-term knowledge, represented by a degree-of-interest value. Overall, our degree-of-knowledge model for a developer assigns a real value to each source code element (i.e., classes, methods and fields) for each developer.

3.3.1 Degree-of-Authorship

From our initial study of nineteen industrial developers (Study_{EXP_DOK}), we determined that a developer’s knowledge of a source code element depends on whether the developer has authored and contributed code to the element and how many changes not authored by the developer have subsequently occurred (code stability). We thus consider the degree-of-authorship (DOA) of a developer in an element to be determined by three factors: first authorship (*FA*), the number of deliveries (*DL*) and the number of acceptances (*AC*).

3.3.2 Degree-of-Interest

The degree-of-interest (DOI) represents the amount of interaction—selections and edits—a developer has had with a source code element [52] as described in Section 3.1.1.

3.3.3 Degree-of-Knowledge

We combine the DOA and DOI of a source code element for a developer to provide an indicator of the developer’s familiarity in that element. We use a linear combination as an initial starting point:

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI$$

3.4 Determining DOK Weightings

Completing our definition of a degree-of-knowledge value for a source code element requires determining appropriate weightings for the factors contributing to the degree-of-authorship and for the degree-of-interest. As there is no specific theory we can use to choose the weightings, we conducted an experiment (Experiment_{DOK}) to determine appropriate values empirically. In essence, the experiment involves gathering data about authorship from the revision history of a project, about interest by monitoring developers' interactions with the code as they work on the project and about knowledge by asking developers to rate their level of knowledge of particular code elements. Using the developer ratings, we then apply multiple linear regression to determine appropriate weightings for the various factors.

We report in this section on an initial determination of weighting values based on the data collected from the seven developers at Site₁. Our intent is to find weightings that serve as a basis to support exploratory investigations of the degree-of-knowledge model. Whether or not the weightings apply across a broader range of development situations is discussed in Section 4.2.4.

3.4.1 Method

At time T_3 in Figure 3.4, we chose, for each developer, forty random code elements that the developer had either selected or edited at least once in the last seven days, or which the developer had first authored ($FA > 0$) or delivered changes to ($DL > 0$) in the last three months. We chose forty as a compromise between gaining data about enough elements and not encroaching too much on the developer's working time. Each developer was asked to assess how well he or she knew each of those elements on a scale from one to five. To help the developers with the rating scale, we explained that a five meant that the developer could reproduce the code without looking at it, a three meant that the developer would need to perform some investigations before reproducing the code, and a one meant that the developer had no knowledge of the code. Although this process meant that sometimes we asked the developer about finer-grained code (a field) and sometimes coarser-grained code (a class), subsequent analysis of the data did not show any sensitivity to granularity.

Through this process, we collected 246 ratings for all seven developers. This value is less than the 280 possible ratings because some of the elements we randomly picked were not Java elements (the authorship and interaction data also included XML, Javascript and other types of code) and the developers stated that they would have difficulty rating them; we therefore ignored these elements.

For this experiment, we consider results to be statistically significant with $p < 0.05$.

3.4.2 Analysis and Results

For our first experimental setting, we applied multiple linear regression to the data collected from the source revision logs and the interaction logs collected as the developers worked. Multiple linear regression analysis tries to find a linear equation that best predicts the ratings provided by developers for the code elements using the four variables: *FA* (first authorships), *DL* (deliveries), *AC* (accepts) and *DOI* (degree-of-interest). Multiple linear regression is suitable for our data, even though the user ratings are ordinal, because we are attempting to find an approximation, not a certain class, for the user ratings.

The values of some of the variables, especially *DOI* and *AC* can be substantially higher than the values of the other variables. To account for these different scales that could potentially make the weighting factors difficult to ascertain, we applied the analysis both with and without taking the natural logarithms of the values. With the developer rating (on a scale of one to five) as the dependent variable, the best fit of the data (using R Square) was achieved with the values presented in Table 3.5, when the natural logarithm of the *AC* and *DOI* values was used. The resulting DOK equation is as follows.

$$\begin{aligned} \mathbf{DOK} = & 3.293 + 1.098 * \mathbf{FA} + 0.164 * \mathbf{DL} \\ & - 0.321 * \ln(1 + \mathbf{AC}) + 0.19 * \ln(1 + \mathbf{DOI}) \end{aligned}$$

Negative values of *DOI* indicate usage that is not recent. In this analysis, we considered any negative *DOI* value to be zero so as to not unduly penalize *DOK*. If we had allowed negative *DOI* values, then elements that the developer had never interacted with may have a higher *DOK* value than elements interacted with long ago.

Table 3.5: Coefficients for Linear Regression

		Weighting	Std. Error	p-value
Intercept	3.293		0.133	< 0.001
<i>FA</i>		1.098	0.179	< 0.001
<i>DL</i>		0.164	0.053	0.002
$\ln(1 + AC)$		-0.321	0.105	0.002
$\ln(1 + DOI)$		0.190	0.100	0.059

The *FA*, *DL* and *AC* variables are significant in this model as well as not correlated and thus help to explain the user ratings. The *DOI* variable is very close to being significant. An analysis shows that the *DOI* is not correlated to any of the other variables. Furthermore, applying linear regression to a model without *DOI* as a variable, results in weighting factors for *FA*, *DL* and *AC* that are very close (on average, 4% change) to the ones from the model using *DOI*. However, a model without *DOI* has a lower “goodness of fit”, R Square, of 0.23 instead of the 0.25 in the other model. These results suggests that *DOI* still plays a predictive role in the model, despite not reaching significance. We hypothesize that the lack of significance is from the lack of elements with a positive *DOI* in the set of randomly chosen elements. Only 7% of all data points have a positive *DOI* whereas 28% have a positive *FA*, 50% have a positive *DL* and 57% have a positive *AC* component. A replication of the experiment presented in Section 4.2.4 confirms our hypothesis. It is not surprising that the weighting factor for *AC* is negative: if a developer Dev_2 changes a code element for which the developer Dev_1 has a high degree-of-knowledge, the Dev_1 ’s knowledge of the element after the change should be lower since he does not know what was changed. This decrease in knowledge is consistent with developer’s comments in *Study_{EXP-DOK}* on code stability (Section 3.1.3).

The F-ratio, a test statistic used for determining the predictive capability of the model as a whole, is 19.6 with $p < 0.000001$. This states that the model based on our four predictor variables has a statistically significant ability to predict the user rating. The overall model has an estimated “goodness of fit”, R Square, of 0.25 (adjusted R Square is 0.23). R Square represents the fraction of the variation in our

user rating that is accounted for by our variables. The correlation coefficient R that represents a measure of the overall fit between our predictor variables and the user rating is 0.50. The standard error of the estimate is 1.17. The 0.25 R Square value shows that our model does not predict the user rating completely. However, the p -value of the overall model as well as the p -values for the variables indicate that there is a statistically significant linear relationship between our model and the user ratings and that each of the four variables contributes to the overall explanation of the user rating.

The threats to the validity of Experiment_{DOK} are further discussed in Section 4.3.

Chapter 4

Evaluation of the DOK Model

To investigate whether the degree-of-knowledge model can be used to improve a developer's productivity and quality of work, we conducted exploratory case studies. In these case studies, we looked at the three questions that were raised by others as important in the literature: who should I ask about a part of the system's code base [56, 60], which changes to the code base should I know about [25, 44] and what code do I need to know about [18, 57]. The results of the case studies provide support that our DOK model can support developers in identifying interesting changes and can improve upon existing approaches for the expertise-finding problem (Section 4.1).

To examine how individual differences across teams affect the factors of the DOK model and their relative effect, we collected data from two more development sites. We found that the factors used in the DOK model are indicative for a developer's knowledge of code regardless of the developer's environment, such as the team, project phase or working style. We also found that, despite their significance, differences in authorship and interaction only have a minor impact on the usage of the DOK model in the expertise-finding study (Section 4.2.4).

4.1 Case Studies

To determine if degree-of-knowledge (DOK) values can provide value to software developers, we performed several exploratory case studies. The first case study

considers the problem of finding experts who are knowledgeable about particular parts of the code. The second case study considers a mentoring situation where an experienced developer might help a new team member become familiar with that part of the code base (i.e., onboard [18]). The third case study considers whether a developer's DOK values can be used to identify which changes to the code might be of interest amongst the many changes occurring during development.

We conducted these case studies with developers at three different sites, Site₁, Site₂ and Site₃, with Site₁ and Site₂ representing different sites of the same company.¹ Site₁ involved the seven developers that we already reported on in Section 3.2. Site₂ involved five professional developers (C1 through C5) who were building a client/server system, using IBM's Rational Team Concert² system as the source repository. The participants had an average of 10 years of professional experience. Site₃ involved three professional developers (E1 through E3), who were working on a closed-source development effort; one developer was working part-time. The participating developers had an average of 2.5 years of professional experience at the time of the case study.

The first two case studies were conducted with the seven developers at Site₁. We repeated the study on finding experts with five developers working at Site₂. The third case study was performed with three developers at Site₃. We conducted these case studies with developers at the three different sites to accommodate for developer's availability and the accessibility to the relevant data as well as for robustness reasons (see more in Section 4.2.4). A discussion of the threats to validity of our case studies is presented in Section 4.3.

4.1.1 Finding Experts

The problem of finding experts is to try to identify which team member knows the most about each part of the codebase. Our degree-of-knowledge model applies directly to this problem since the developer with the highest DOK value for an element across team members is likely to be the one with the highest expertise.

¹The site numbers used are different from the ones presented in our previous work [32].

²The subjects used RTC version 2.0, and Eclipse 3.4 and 3.5.

Method

We conducted this case study at two sites. The seven developers at Site₁ have a strong model of code ownership with code split amongst team members and certain individuals responsible for certain packages. At Site₂, the developers have a model of mutual code ownership with team members often working on the same code. Due to the differences in the code ownership style, we performed the case study with slight adaptations at each site.

At both sites, the code is partitioned into projects, where a project is a logical group of Java packages. At Site₁, where there is a strong model of code ownership, we chose two projects with which most members of the team had interacted. One project comprised 21 Java packages; the other comprised 88 packages. For each class in these packages, and for each of the seven developers participating in our study, we calculated the DOK value for each class-developer pair and then computed DOK values for each package-developer pair by summing the developer's DOK values for each class in the package. For calculating the DOK values, we used the interaction data over the past seven work days and the authorship information over the past three months—three months was the most history available for these elements at this site due to a major porting of the code three months prior.

Using these DOK values, we produced a diagram, which we call a knowledge map, that showed for each package in a project, the developer with the highest DOK for that package. Figure 4.1 presents a part of the knowledge map for one project.³ In this figure, each developer is assigned a color and each package is colored (or labeled) according to the developer with the highest DOK values for that package. For one project, 17 of the 21 packages (80%) were labeled and for the second, 61 out of 88 (69%) were labeled. Thus, 78 packages in total were labeled. For the remaining 31 packages, the DOK values for all developers were not positive, meaning primary expertise might lie outside the team.

We then conducted individual sessions with each of the seven team members. In each session, we first showed the developer a list of the packages without any DOK values indicated and asked the developer to write down the name of the team member whom he thought knows the package the best. When requested,

³Please note this figure is best viewed in color.



Figure 4.1: Part of a Knowledge Map

we showed a developer a list of the classes within a package. After gathering this data, we showed the developer the knowledge map and asked if the map reflected his view of which developer knows which part of the code.

At Site₂, due to the mutual code ownership amongst the developers of this team, we focused on the class level rather than package level. We again chose two projects with which most members of the team had interacted. From these two projects, we chose 96 classes with which most members of the team had interacted. These 96 classes were spread over 16 packages. For each class and for each of the five developers participating in our study, we then calculated the DOK value for each class-developer pair. This time, due to its availability, we used the past six months of authorship information and the past seven days of interaction data. We then produced a knowledge map that showed for each class, the developer with the highest DOK for the class. 89 out of the 96 classes (93%) were labeled by our approach. For the remaining 7 classes, our approach did not determine a single

expert due to multiple developers having the same DOK value for the same class. As with the developers at Site₁, we conducted individual sessions with each of the five team members and asked them for each class to determine the member whom he thought knows the class best. After gathering this data, we again showed the developer the knowledge map and asked if the map reflected his view of which developer knows which part of the code.

At both sites, this approach runs the risk of developers over-inflating their expertise in a package/class to avoid not appearing as an expert in anything. We mitigated this risk because the packages and classes we looked at represent only a small fraction of the entire system so a developer who did not indicate expertise in the packages that were part of the case study might still be expert in some other part of the code.

Results from Site₁

We gathered data from six developers (D1-D6); one developer (D7) did not interact with any of the code in the two projects and thus was not able to provide meaningful data.

For the 78 of the 109 packages labeled with a single developer, we gathered 468 (6 developers times 78 packages) assignments from the developers participating in the study. In 301 of these cases (64%), the developers in the study assigned one developer as being the one that “knows the most” or “owns” the package. In 166 of these 301 cases (55%), the result we computed based on DOK values was consistent with the assignments by the developers.

The 55% accuracy value is a lower bound of our approach’s performance given that the developer assignments were sometimes guesses; after seeing the knowledge map the developers realized their assignments were likely wrong. All six developers stated that the knowledge map was reasonable, using phrases like it is “close” (D4) and it “reflects [reality] correctly” (D2).

For the 31 out of the 109 packages for which we did not find anyone using the DOK values, the six developers assigned someone to a package in 104 cases. In 48 of these cases (46%), the packages had not been touched for a number of months and were created six months ago. Given that our DOK values were based on three-

months of data, we were missing the initial authorship data. Developers stated that in “blank cases” (D4) where our DOK did not determine anyone, we should adapt the DOK to go back further in time.

Results from Site₂

For the 89 of the 96 classes labeled with a single developer, we gathered 450 (5 developers times 90 classes) assignments from the developers participating in the study. In 352 of these cases (78%), the developers assigned one developer as being the one that “knows the most” of the class. In 190 of these 352 cases (54%), the result we computed based on DOK values was consistent with the assignments by the developers.

The 54% accuracy value is again a lower bound of our approach’s performance and developers stated that the knowledge map “makes sense” (C1), “seems good” (C3) and “seeing this now, I think I should have put this person’s name [(the one found by our approach)] by some of these [classes]” (C5). They also commented on the fact that for some classes that are very big it is difficult to assign one single developer that knows the most (C2).

Comparison to Expertise Recommenders

For this task, it is possible to compare to other approaches, since earlier work in expertise recommenders has considered the problem of finding experts. As described in Section 2.1.1, these approaches are based solely on authorship information and do not consider interaction data and the ebb and flow in the knowledge. To approximate the results of these earlier approaches at Site₁, we computed experts for each package by summing up all first authorship and delivery events from the last three months for a developer for each class in the package. At Site₂, we used the six months of authorship information to compute experts for each class by summing up all first authorship and delivery events. The developer with the most “experience atoms” [63]—the most events—for a package/class is the expert.

We applied this expertise approach to the two projects at Site₁. In 21 out of the total 109 packages, the expertise approach identified a different expert developer than our DOK-based approach. For these 21 packages, we gathered 69 assign-

ments, in which one of the six developers assigned one developer as being the one that “knows the most” of the package. In 34 of these 69 cases (49%), our DOK-based approach agreed with the developer assignments, whereas the expertise approach agreed in only 17 (24%) of these cases. Thus, the DOK approach improves the results for the packages that were labeled differently by the two approaches by more than 100% and the overall result by 11%.

We also applied this expertise approach to the 96 classes at Site₂. In only 3 of these 96 classes did the expertise approach provide a different result than our approach. This low number of differences likely stems from focusing on the class level at Site₂ compared to the package level at Site₁. At Site₁, in 72% of the cases (79 of the 109 packages under consideration) had more than one person working on classes in the package, resulting in more than one possibility for who is expert for the classes in that package. At Site₂, in only 34% of the cases (33 of the 96 classes under consideration) had more than one person working on the same class. It is in these cases where there is more than one author that one would see the differences appearing between our approach and the expertise approach given our fine-grained modeling of how programmers are mutually changing the class (or package). For the 3 classes in which our approach differed from the expertise approach, we had 9 assignments from the five developers. In 3 of these 9 assignments (33%), our DOK-based approach agreed with the developer assignments, whereas the expertise approach agreed with none of the developer assignments (0%). The agreement between our approach and the developers in the 3 cases stems from weighting first authorship actions differently from delivery events and by modeling the ebb and flow in a developer’s knowledge explicitly via acceptance of changes to the code.

Overall, these comparisons shows that DOK values can improve on existing approaches to finding experts.

4.1.2 Onboarding

Becoming productive when joining a new development project requires learning the basic structure of the code base. The process of becoming proficient with a codebase is known as onboarding [18]. In this case study, we investigated whether DOK values computed from developers with experience in a part of a codebase

could be used to indicate which code elements a newcomer should focus on when trying to learn that part of the code base.

Method

For this study, we randomly chose three developers (D1, D3, D5) at Site₁. We chose developers from Site₁ due to their availability. We asked each developer to describe which code elements from the areas in which he was working would likely be the most useful for a newcomer trying to come up-to-speed on the code. We then generated, for each developer, the twenty elements with the highest DOK and asked the developer to comment on whether these twenty elements would likely be helpful for a newcomer.

Results

Only 2 of the 60 (3%) elements generated across all three developers were considered by the developers to likely be helpful for a newcomer. The other 58 (97%) elements were described by the developers as not being essential for understanding the code. The elements generated using the DOK values were, “only implementations” (D1) or “secondary consumers” (D3). The developers described that a newcomer only needs to understand basic patterns (D1, D5) and that while the elements generated using DOK could serve as examples, it would be necessary to traverse up the inheritance hierarchy to locate the elements the newcomer should study (D1). These comments are consistent with the description of the developers that they often recommend newcomers to look at API elements. The DOK values for the API elements were either very low or zero as they were neither changing nor were they referred to frequently by the developers who authored them. Perhaps the developers had internalized these APIs and did not need to refer to them.

For onboarding, then, the elements with high DOK values were not considered helpful. However, the developers comments suggest that the elements with high DOK might be used as a starting point to find useful elements for onboarding by following call or type hierarchies. These results reflect what developers stated in our initial study on the role of program elements (Section 3.1.3)—the more important the role of an element in the program, the better the knowledge about it.

We leave the investigation of this potential use of DOK values to future research.

4.1.3 Identifying Changes of Interest

In many development projects, keeping up with how the work of teammates might effect one's work typically requires monitoring the progress of changes. In many projects, changes to the source code are tied to bug reports either by listing the bug report in the change or by attaching the change to a bug report (e.g., by attaching the change itself or meta-information such as a task context [52]). By monitoring bug reports instead of inspecting individual source code changes, a developer can be provided more rationale about a change.

Many bug reports for the project change daily. A developer who wishes to monitor changes of interest typically performs searches over the bug repository to determine these changes. Assessing which changes matter and which are irrelevant can be difficult for a developer.

In this case study, we investigated whether a developer's DOK values can be used to select changes of interest to the developer because of overlap between the source code change and the developer's DOK model.

Method

For this study, we computed a DOK model for each of three developers from Site₃. On a particular day, we determined all bugs that had changed in the previous seven days; we refer to this set as B_C . As these three developers work on multiple projects and we are analyzing changes of interest for one project, we used seven days to capture a sufficient amount of change to the project we were targeting. The same seven days were used for developers E1 and E2; a different seven days were used for developer E3. These dates differed to accommodate developers' schedules.

We then determined the subset of bugs that had change information attached to the bug; we refer to this set as B_{CI} .⁴ For each bug report with change information in B_{CI} , we computed the aggregate DOK value for each element in the change information based on the developer's DOK model. We formed the set of bugs with

⁴For this project, the change information were task contexts collected automatically as a developer worked and thus represented both the elements changed in the revision system and elements considered by the programmer in making the change.

Table 4.1: Size of Bug Recommendation Sets

Developer	B_C	B_{CI}	B_{POT}	B_R
E1	123	26	20	3
E2	123	26	7	2
E3	76	28	5	1

an aggregate DOK value that was positive; we refer to this set as B_{POT} . We then removed the bugs in B_{POT} when the developer was already mentioned on the bug as an assignee, reporter or had commented on the bug. The resulting set of bugs are those we report as bugs of interest to the developer; we refer to this set as B_R . For each bug in B_R , we asked the developer whether they had read the bug or whether they would have wanted to be aware of the bug.

Results

Table 4.1 summarizes the number of bugs in each set for each developer. Developer E1 was asked about the relevance of three bugs. He noted that he had read each of these bugs to make sure no further action was required on his part. Developer E2 was asked about two bugs. She noted that for one of these bugs, she was asked in person about the contents of the bug although she had not read it previously. The developer was impressed our approach had picked it out of the many bugs that were undergoing change. Developer E3 was recommended one bug. He had read the bug but did not comment explicitly about whether the bug was relevant to his work.

Overall, our approach provided relevant information to developers in four out of six cases by recommending non-obvious bugs based on the developers' DOK values. Unfortunately, it was not possible to assess false negatives in this study. Although our case study used a very coarse-grained metric to determine relevance of bugs using DOK values, we were able to easily recommend more relevant bugs than noise.

4.1.4 Case Studies Summary

The case studies we conducted supply initial evidence that our model can provide value in scenarios such as expert finding and identifying changes of interest. The case studies also show that our model does not adequately reflect a developer's knowledge for API elements. We discuss how our approach for approximating DOK can be improved in Section 7.1.3. This initial evidence suggests that further study of the DOK model is warranted. In particular, the weightings for the factors contributing to the DOK model and the appropriate amounts of data to use to compute DOK values require experimentation with more developers in a greater variety of situations.

4.2 Robustness of the Model

Our case studies provide initial evidence that the DOK model can help developers determine who to ask about parts of the code base and of which changes they should be aware. These case studies leave open the question of whether the weightings for the factors contributing to the DOK model are robust over different development sites, teams and situations. A DOK model tailored specifically towards the developers at Site₂ might, for example, provide better results for the expert finding case study we performed at that site. To investigate the robustness of our model and how differences could influence the case studies, we collected data from five developers (C1 through C5) at Site₂ and two developers at Site₃ (E1 and E2).

In the following, we report on the differences in authorship and interaction behavior between the sites and the effect that these differences have on the weighting factors. We also revisit the expert finding case study and look at how a DOK model based on weighting factors tailored specifically towards Site₂ performs.

4.2.1 Differences in Development Teams

We gathered data from development teams at Site₂ and Site₃ to compare it to the data we gathered from Site₁ and presented in Section 3.2. Due to accessibility to the data and developer's availability, the developers from Site₃ were working at the same site, but are different to the ones reported on in the previous section (Section 4.1). The team at Site₃ involved two professional developers (both male),

who build open source frameworks for Eclipse as part (but not all) of their work and who use CVS⁵ as their source repository system. One developer had three years of professional experience, the other had five years.

Authorship

Both sites, Site₂ and Site₃, show a lower rate of change in authorship information than Site₁. Over a period of six weeks, developers at Site₂ have a median number of 1890 accept events (range: 1176 to 2781) and developers at Site₃ a median of 2081 (range: 1186 to 5710). These numbers are lower than the ones at Site₁ (median of 62500), but even at the lower rate, a substantial amount of the code in a developer's environment is changing each day.

For developers at Site₂, the rate of first authorship events (median: 53.5; range: 0 to 208) and delivery events (median: 121.5, range: 17 to 315) is also lower than for developers at Site₁ with a median of 323 for first authorship and 230 for delivery events. Developers at Site₃ had a higher rate of first authorship (median: 398.5, range: 78 to 2312) and delivery events (median: 602, range: 157 to 2712) than developers at Site₁. Both sites, Site₂ and Site₃, show similarly high variations as Site₁ across and within developers with respect to first authorship and delivery events. This can also be seen in the differences in the means and the high standard deviations presented in Table 4.2.

There are several potential reasons for the differences in the numbers between developers at different sites. First, the code at Site₂ and Site₃ is smaller and is being worked on by a smaller team, potentially causing a different event profile, in particular causing less accept events to occur. Second, there are individual factors such as different models of code ownership (see Section 4.3). Third, different teams were in different project phases over the period of time for which we collected the data. Finally, whereas the source repository system in use at Site₁ and Site₂ supported atomic change sets with explicit accept events occurring within the development environment, at Site₃, the source revision system lacked both of these features. Instead, we inferred delivery events based on revision information to source code elements; if a developer performed several commits to the revision

⁵www.nongnu.org/cvs, verified 19/11/10

Table 4.2: Average of Six Weeks of a Developer’s Authorship Data Over a Three Month Period (FA = First Authored, DL = Delivered, AC = Accepted, Change stands for the change in elements per week; subjects C1 to C5 are from Site₂, subjects E1 and E2 from Site₃)

Subj.	FA	# Events DL	AC	# Distinct Elements	
				FA & DL	Change
C1	46 (± 33)	71 (± 29)	1945 (± 510)	84 (± 51)	16% ($\pm 18\%$)
C2	98 (± 45)	225 (± 62)	1836 (± 563)	212 (± 48)	10% ($\pm 16\%$)
C3	7 (± 8)	32 (± 20)	2134 (± 518)	24 (± 27)	23% ($\pm 33\%$)
C4	63 (± 22)	118 (± 27)	1885 (± 505)	121 (± 33)	17% ($\pm 14\%$)
C5	157 (± 70)	271 (± 62)	1879 (± 324)	205 (± 56)	10% ($\pm 13\%$)
E1	225 (± 128)	349 (± 141)	3388 (± 1374)	446 (± 237)	16% ($\pm 21\%$)
E2	1035 (± 802)	1262 (± 890)	1665 (± 315)	1807 (± 1496)	9% ($\pm 9\%$)

system as part of one logical change, we record this as multiple delivery events. The lack of an explicit accept event that could be logged meant that we had to infer at the end of each day that all outstanding changes were accepted, potentially increasing the accept events.

The change per week in the unique elements a developer first authored or delivered changes to, is again low with 15% at Site₂ and 9% at Site₂, providing further evidence that authorship captures a longer-term component of knowledge.

Interaction

Table 4.3 presents the interaction data from Site₂ and Site₃. The median number of interaction events each developer had over the last seven work days, is in a similar range at all three sites. Developers at Site₂ had a median of 8594 (range: 1065 to 20291) interactions over seven working days, developers at Site₃ had a median of 6191 (range: 2151 to 10152) interactions over seven working days, compared to a median of 7823 (range: 3296 to 12841) interaction events at Site₁. Different to the other two sites, developers at Site₂ interacted with relatively few distinct elements, with a median of only 227 (range: 58 to 634), despite to having the most interaction events on average. In comparison, developers at Site₃ interacted with a median of 887 (range: 446 to 1271) distinct elements and developers at Site₁ with

Table 4.3: Developer’s Interaction Data Averaged Over Five Work Days (subjects C1 to C5 are from Site₂, subjects E1 and E2 from Site₃)

Subj.	# Interaction Events (over 7 days)	# Distinct Elements (over 7 days)	# Els with pos. DOI (per day)	Change in pos. Els (per day)
C1	8867 (± 522)	210 (± 13)	35 (± 7)	19% ($\pm 19\%$)
C2	2467 (± 1332)	96 (± 39)	17 (± 10)	54% ($\pm 37\%$)
C3	17596 (± 2772)	419 (± 46)	21 (± 0)	0% ($\pm 0\%$)
C4	17654 (± 2701)	552 (± 99)	28 (± 4)	11% ($\pm 13\%$)
C5	7037 (± 836)	232 (± 27)	24 (± 4)	34% ($\pm 26\%$)
E1	8459 (± 1747)	987 (± 248)	60 (± 13)	60% ($\pm 9\%$)
E2	3931 (± 2075)	695 (± 285)	61 (± 20)	61% ($\pm 19\%$)

a median of 898 (range: 514 to 1651). When looking at the daily change of the set of elements with a positive DOI, the change at Site₃ is with 61% almost as high as at Site₁ (68%), whereas, on average, only 23% of the elements changed per day at Site₂.

The differences in interactions between individuals is again quite substantial as it depends on factors such as the individual’s role on the team, the codebase size, and what the developer was working on over the past seven days. For example, one developer (C3) was gone for several days causing the change in elements to be zero. Furthermore, developers at Site₂ were working for parts of the data collection in a testing phase, which might have caused the lower number of distinct elements they were interacting with. The lower number in overall interaction events at Site₃ might be due to the developers working only part-time on the project. We discuss these issues further in Section 4.3.

4.2.2 One Model for All

To determine how different teams affect the weightings initially computed at Site₁, we conducted an experiment similar to the one described in Section 3.4 with the teams from Site₂ and Site₃. As we did at Site₁, we again chose forty random code elements for each developer with the same characteristics as at Site₁—the developer had either selected or edited the code element at least once in the last seven

Table 4.4: Coefficients for Linear Regression over all Sites.

		Weighting	Std. Error	p-value
Intercept	3.223		0.081	< 0.001
<i>FA</i>		0.962	0.134	< 0.001
<i>DL</i>		0.213	0.037	< 0.001
$\ln(1 + AC)$		-0.273	0.068	< 0.001
$\ln(1 + DOI)$		0.270	0.049	< 0.001

days, or first authored or delivered changes to in the last three months. We asked each developer to score the presented elements from one to five based on how well he or she knew each of these elements. In total, we collected 525 ratings for all fourteen developers. We applied multiple linear regression to the data, again using the four variables *FA* (first authorships), *DL* (deliveries), *AC* (accepts) and *DOI* (degree-of-interest) and applying the analysis for all possible combinations with and without taking the natural logarithms of the values. For this experiment, we took into account three months of authorship and seven work days for interaction for all three sites. The best fit of the data to the developer rating (dependent variable) was achieved with the values presented in Table 4.4 using the natural logarithm for *AC* and *DOI* resulting in the following DOK equation:

$$\begin{aligned} \mathbf{DOK} = & 3.223 + 0.962 * \mathbf{FA} + 0.213 * \mathbf{DL} \\ & - 0.273 * \ln(1 + \mathbf{AC}) + 0.270 * \ln(1 + \mathbf{DOI}) \end{aligned}$$

Using all data, all four variables are now significant in the model, the F-Ratio is 30.9 with $p < 0.000001$ (F-Ratio for Site₁ was only 19.6) and the standard error of the estimate 1.19. This states that our model based on the four predictor variables has a statistically significant ability to predict the user ratings. The R Square is 0.19 (adjusted R Square is 0.19), lower than the 0.25 R Square for the linear regression taking into account only Site₁. The 0.19 R Square indicates that our model does not predict the user rating completely and the fact that it is even lower than before suggests that there might be site specific factors. One reason that might also affect the results is that in the 525 randomly picked data points, we now have 11% (before only 7%) with a positive *DOI*, 20% (before 28%) with a positive *FA*, 47% (before 50%) with a positive *DL* and 48% (before 57%) with a positive *AC*.

Overall, the weighting factors for all four variables changed, on average, by only 25% from our initial model. The weighting factor for *FA* changed from 1.098 to 0.962, *DL* from 0.164 to 0.213, *AC* from -0.321 to -0.273 and *DOI* from 0.190 to 0.270. Given that a developer, on average, only has 1.4 ($SD = 0.4$) first authorship or delivery events for each element he is making changes to and accepts 2.0 ($SD = 0.4$) changes for each element he accepts changes for over a period of six weeks, the changes in the authorship variables appear relatively minor. Considering the vast differences in authorship and interaction behavior between the different sites and developers, this suggests that our initial model can predict DOK values with reasonable accuracy, even when the environments from which the data was collected have different profiles.

4.2.3 A Site-Specific Model

Despite the small differences between our initial model and the model for all sites combined, there are quite substantial differences in the authorship and interaction behavior between different sites. A DOK model tailored specifically to developers from one site might substantially improve upon our initial model. Therefore, we applied multiple linear regression to the data that we gathered from Site₂.⁶ The best fit was achieved with the values presented in Table 4.5 resulting in the following DOK equation:

$$\text{DOK} = 2.919 + 0.749 * \text{FA} + 0.286 * \text{DL} \\ - 0.044 * \text{AC} + 0.353 * \ln(1 + \text{DOI})$$

This time, the weighting factors differ, on average, by 70% from our initial model. In particular, the weighting factor for *DOI* is substantially higher, with the variable being significant, whereas the weighting factor for first authorship is lower. Also, the best fit was achieved without taking the natural logarithm of *AC* and *AC* is only close to being significant in this model. These differences are not surprising considering the lower rate of authorship events and the therefore relatively high rate of interactions. The F-Ratio in this case is 11.0 with $p < 0.000001$, a standard error of 1.23, and an R Square of 0.19 (adjusted R Square of 0.17).

⁶Due to the small number of developers available at Site₃, we chose to only create a site-specific model for Site₂.

Table 4.5: Coefficients for Linear Regression for Site₂.

		Weighting	Std. Error	p-value
Intercept	2.919		0.120	< 0.001
<i>FA</i>		0.749	0.275	0.007
<i>DL</i>		0.286	0.082	< 0.001
<i>AC</i>		-0.044	0.023	0.060
$\ln(1 + DOI)$		0.353	0.062	< 0.001

To evaluate the impact of a site-specific model, we applied the Site₂-specific DOK model to the expert finding case study we conducted at Site₂ and compared it to the results we obtained by using our initial model. It turns out, that using the new weighting factors, all assignments stay the same as the ones computed using our initial model. One potential reason for this result is that the developers did not have a lot of interactions with the classes under consideration over the past seven days. The result also suggests that our initial model can be used to predict reasonable DOK values in different environments.

4.2.4 Robustness Summary

Despite big variations in authorship and interaction behavior between different development sites and even within development teams, our model provides value in case studies across developers and different environments. Our initial model is relatively stable when adding more data points and performs equally well as a site-specific DOK model in the expert finding case study. This evidence suggests that our model has a certain robustness and can be beneficial independent of a particular site. However, even though we chose three different teams from different locations and working on completely different projects, future work should include further studies in a greater variety of situations and over different project phases of teams.

4.3 Threats to Validity

The degree-of-knowledge (DOK) model is influenced by both the software development process and the software system being developed. We detail a number of the factors influencing DOK and how they pose threats to the validity of the experiments and studies we conducted.

4.3.1 Amount of Data

Our experiments on determining the weighting factors were based on three months of authorship data and seven working days of interaction data. We chose this duration for authorship data based on interviews in Study_{EXP.DOK}. In these interviews, developers had suggested three months as a lower bound for the period of time in which one still has knowledge about code after authoring it. Also based on our initial study, we chose seven working days of interaction data. Seven working days was the average number of working days that showed a significant result for the correlation between a developer’s knowledge and his interaction.

4.3.2 Multiple Stream Development

The seven developers we studied at the first site share code in streams, which are similar to branches in a source revision system. The developers deliver their changes to one or more streams and accept changes from streams into their workspace. Normally, the developers we studied work only on a small number of streams. However, during our data collection and study period, some of the developers were working on many streams: “it’s not a normal situation, right now [it is] very branched out, [and] I almost spend more time merging than working on it” (D5). When streams representing different versions of the same code are merged, additional authorship events are recorded that could skew the results of our experiment and studies. We tried to minimize the influence of these extra events by focusing on only one major stream for each developer.

4.3.3 Project Phase

Developers interact differently with a codebase depending on the phase of the project on which they are working. In the week in which we collected interaction

data at Site₁ to determine the DOK weightings, the team was in a testing phase for an upcoming milestone release. Some of the developers reported that they were only performing minor adjustments to the code but not really making any bigger changes to ensure the code did not break. Some developers stated that a couple of months before they were working on new features, during which a substantial amount of new code was created and the focus of individual developers in a part of the codebase was higher.

The number and size of change sets and the tasks of the developers (i.e., testing versus feature development) influences the authorship and interaction data. By taking into account three months of authorship data, seven days of interaction data and also confirming the results of the DOK weighting experiment at a second site, we have tried to minimize the impact of project phases on our results. Further longitudinal studies are needed to better understand the impact of project phases on indicators such as DOK.

4.3.4 Individual Factors

As previously discussed, different teams have different styles of code ownership, varying from individual ownership of whole packages to mutual ownership for each class. The style of code ownership within a team influences the data input to determine DOK values. We have tried to mitigate the risk of these different styles by considering for one scenario whether the weightings determined for one team applied to another team.

A developer's activity also has an influence on the data. For instance, one developer in our study was working on more than four different streams and was expending effort that week merging streams together. When we applied linear regression on the data points gained from only this developer, the result was not significant. For other developers, the goodness of fit of the model is more than twice as good as the goodness of fit for all developers. Thus, while individual factors, such as the team's style of code ownership and activities of individuals influence results, by having several developers, each with a different activity, we have tried to minimize the risk of individual biases.

Chapter 5

The Information Fragments Model

Each day, a developer faces questions over multiple kinds of information, such as work items, documentation and web feeds, to name just a few. To support a developer in integrating information according to his individual preference to answer this type of questions, we developed the information fragments model. This developer-centric model allows a developer to automatically compose relevant portions of information and to easily choose how to display the composed information.

In an exploratory study, Study_{EXP_FR}, we investigated the range of questions that span across multiple kinds of information and that our model should support. Section 5.1 presents the 78 questions we identified in the study. We then discuss the difficulties in answering these types of questions with existing approaches (Section 5.2), before presenting the information fragments model in Section 5.3.

5.1 Developers' Questions

To determine the specific questions developers need answered that span different kinds of information, we interviewed eleven experienced software developers. We learned about 78 questions that the developers face and about the differences in the interpretation of very similar questions. To ease presentation, we use the term *domain* in the remainder of the dissertation to refer to information of a particular

kind, such as the domain of work items or the domain of source code.

5.1.1 Subjects and Interview Process

We conducted open interviews with eleven professional developers from three different sites of one company. These developers represented a spectrum of roles and experience. The roles ranged from junior developer to team leads. The experience of these developers ranged from 1 to 22 years.

In a pilot for this study, we asked developers directly about questions requiring information from multiple domains that occur for them during development. Through the pilot, we found that the developers had difficulty understanding the kinds of questions that might meet this criteria.

Thus, we changed the interview method to start with a brief demonstration of a small tool we built that enabled the composition of source code, change sets, work items and team information. Our intent in showing this prototype was to stimulate developers to think about questions that might arise when such composition is possible. We asked the developer to describe such questions and how he would want them answered. Throughout the interview, we adapted our questions to the scenarios the developer described.

Each interview session was between 15 and 60 minutes depending on the developer's availability and responsiveness. Throughout each session, the interviewer (the author of this dissertation) took handwritten notes. We parsed our notes looking for the questions developers stated, as well as the meaning of the questions to the developers.

5.1.2 Interview Results

From the eleven interviews, we determined 78 questions that span across multiple domains. Only one of the 78 questions was stated by two different developers; the other 77 questions were each stated by only one of the developers. Table 5.1 lists all 78 questions. These questions span eight domains of information: source code (SC), change sets (CHS), teams (T), work items (WI), web sites and wiki pages (WW), comments on work items¹ (CO), exception stack traces (ST) and test cases

¹Based on the developers' statements, we considered emails as being equivalent to comments.

(TC). Table 5.1 shows which domains, based on developer statements, are needed to answer each question. For ease of reading, we have grouped the questions into categories that roughly correspond to the domains needed to answer the questions. Some questions are annotated with a *, which denotes questions explicitly stated by at least one developer. All other questions are interpretations we have made as the interviewed developer either gave a long statement to describe the scenario or more context was needed to be able to state the question. The majority of the questions (51 of the 78) are based on the domains we presented in our demonstration at the start of the interview; the remaining questions incorporate a domain we did not present.

Table 5.1: Developers' Questions and the Operators and Domains for Desired Answers
 (*: question explicitly stated by a developer, *id*: identifier matching, *t*: text matching)

Question	Operator	Source Code	Change Sets	Teams	Work Items	Comments	Web/Wiki	Stack Traces	Test Cases
<i>Who is working on what (people specific)</i>									
1. Who is working on what?*	<i>id</i>	X	X	X	X				
2. What are they [coworkers] working on right now?*	<i>id</i>	X	X	X	X				
3. What have other people been working on?*	<i>id</i>	X	X	X	X				
4. How much work [have] people done?*	<i>id</i>	X	X	X	X				
5. Who changed this [code], focused on person?*	<i>id</i>	X	X	X	X				
6. Who to assign a code review to? / Who has the knowledge to do the code review?	<i>id</i>	X	X	X	X				
7. What [have] people done lately?*	<i>id</i>	X	X	X	X				
8. Who is working on what at the moment?*	<i>id</i>	X	X	X	X				
9. What has [a particular team member] been doing?*	<i>id</i>	X	X	X					
10. What have people been working on?*	<i>id</i>			X	X				
11. Which code reviews have been assigned to which person?*	<i>id</i>			X	X				
12. Who to assign a code review to? / Who has time for a code review?	<i>id</i>			X	X				
<i>Changes to the code (code specific)</i>									
13. What is the evolution of the code?	<i>id</i>	X	X	X	X				
14. Why were they [these changes] introduced?*	<i>id</i>	X	X	X	X				
15. Who made a particular change and why?	<i>id</i>	X	X	X	X				
16. What classes has my team been working on?*	<i>id</i>	X	X	X	X				
17. What are the changes on newly resolved work items related to me?	<i>id</i>	X	X	X	X				
18. Who is working on the same classes as I am and for which work item?	<i>id</i>	X	X	X	X				
19. Who changed this [code], focused on code?*	<i>id</i>	X	X	X	X				
20. What is the whole history of this file?	<i>id</i>	X	X	X	X				
21. What has been happening on [this] class?*	<i>id</i>	X	X	X	X				
22. What [have] people changed lately?*	<i>id</i>	X	X	X	X				
23. What changes have been made and why?*	<i>id</i>	X	X		X				
24. What has changed between two builds [and] who has changed it?*	<i>id</i>	X	X	X					
25. Who has made changes to my classes?	<i>id</i>	X	X	X					
26. Who is using that API [that I am about to change]?*	<i>id</i>	X	X	X					
27. Who created the API [that I am about to change]?*	<i>id</i>	X	X	X					
28. Who owns this piece of code? / Who modified it the latest?*	<i>id</i>	X	X	X					

Table5.1: (Continued)

Question	Operator	Source Code	Change Sets	Teams	Work Items	Comments	Web/Wiki	Stack Traces	Test Cases
29. Who owns this piece of code? / Who modified it most?*	<i>id</i>	X	X	X					
30. Who to talk to if you have to work with packages you haven't worked with?	<i>id</i>	X	X	X					
31. How much has changed [in the project code]?*	<i>id</i>	X	X	X					
32. [Is anyone] intending to commit anything to that class?*	<i>id</i>	X	X	X					
33. Where have changes been made related to you?*	<i>id</i>	X	X	X					
34. Who is responsible for this code? (Who made the latest change?)	<i>id</i>	X	X	X					
35. Which team is responsible for this code? (Who has made most changes to the code?)	<i>id</i>	X	X	X					
36. What classes have been changed?*	<i>id</i>	X	X						
37. [Which] API has changed (to see which methods are not supported any more)?*	<i>id</i>	X	X						
38. What's the most popular class? [Which class has been changed most?]*	<i>id</i>	X	X						
39. Which other code that I worked on uses this code pattern / utility function?	<i>id</i>	X	X						
40. Which code has recently changed that is related to me?	<i>id</i>	X	X						
41. How do recently delivered changes affect changes that I am working on?*	<i>id</i>	X	X						
42. What code is related to a change?	<i>id</i>	X	X						
43. Where has code been changing [this week]?*	<i>id</i>	X	X						
44. Which classes have been changed between two builds?	<i>id</i>	X	X						
45. What is going on [in a package]?*	<i>id</i>	X	X						
46. [Which] changes [have been made] between these days or after this day?*	<i>id</i>	X	X						
47. What classes in this component were modified since version [...]?*	<i>id</i>	X	X						
<i>Work item progress</i>									
48. What is the recent activity on a plan item?	<i>id</i>				X	X			
49. Which features and functions have been changing?*	<i>id</i>				X	X			
50. Has progress been made on blockers (blocking work items) in your milestone?	<i>id</i>				X	X			
51. Which work items/plan items are most active?	<i>id</i>				X	X			
52. How active is the [plan item]? [How many comments were made on related work items]?*	<i>id</i>				X	X			
53. Are there any new comments on interesting work items?	<i>id</i>				X	X			
54. [What are the] emails related to line items and defects that are features?*	<i>t</i>				X	X			
55. Which work item has recently changed that is related to me?	<i>id</i>			X	X	X			

Table5.1: (Continued)

Question	Operator	Source Code	Change Sets	Teams	Work Items	Comments	Web/Wiki	Stack Traces	Test Cases
56. What are the comments on newly resolved work items that are related to me?	<i>id</i>			X	X	X			
57. Is progress (changes) being made on plan items?	<i>id</i>		X		X				
58. What is the activity on a line item (feature)?	<i>id</i>		X		X				
<i>Broken builds</i>									
59. What caused this build to break? (Which change caused the stack trace?)	<i>id</i>	X	X					X	
60. What has caused this build to break? (look at stack trace and intersect with change sets)	<i>id</i>		X					X	
61. Who caused this build to break? (Who owns the broken tests?)	<i>id</i>	X	X	X					X
62. Who changed the test case most recently that caused the build to fail?	<i>id</i>	X	X	X					X
63. Which changes caused the tests to fail and thus the build to break?	<i>id</i>	X	X					X	X
<i>Test cases</i>									
64. Who owns a test case? (Who resolved the last work item that fixed the test case?)	<i>id</i>	X	X	X	X				X
65. Who is responsible for a failing test case? (stack trace)	<i>id</i>	X	X	X				X	X
66. How do test cases relate to work items?	<i>id</i>	X	X		X				X
67. How do test cases relate to packages/classes?	<i>id</i>	X							X
<i>References on the web</i>									
68. Which API has changed (check on web site)?	<i>t</i>	X					X		
69. [Is an entry] in newsgroup forum addressed to me because of the class mentioned?*	<i>t</i>	X					X		
70. What is coming up next week [for my team]? [What is my team doing?]*	<i>t</i>			X			X		
71. What am I supposed to work on [plan on wiki]?*	<i>t</i>			X			X		
72. Who has to do what? [team activity]*	<i>t</i>			X			X		
<i>Other Questions</i>									
73. How is the team organized?*	<i>id</i>	X	X	X					
74. Who has made changes to [a] defect?*	<i>id</i>		X	X	X				
75. Who has made comments in [a] defect?*	<i>id</i>			X	X	X			
76. [What is] the collaboration tree around a feature?*	<i>id</i>			X	X	X			
77. Which conversations in work items have I been mentioned [in]?*	<i>t,id</i>			X	X	X			
78. What are people commenting [on] all work items I am involved with?*	<i>id</i>			X	X	X			

In our interviews, we focused on the variety and richness of questions rather than their frequency. From the interviews, we determined that some of these questions are considered a couple of times a day, such as “What classes have been changed?”(36). On other questions, developers spend a lot more time. One developer stated that he spends 70% of his time on the question “Which conversations in work items have I been mentioned [in]?”(77) to make sure he does not block other developers from working. A more extensive field study is needed to measure exactly how often each of these questions arises throughout a work day or week, or how much time a developer spends on each of these questions.

Some of the questions sound very similar in their wording, but their answer differs depending on the individual interpretation of the developer. For example, even though the two questions “Who is working on what?”(1) and “What have people been working on?”(10) seem very similar, the answer to question (1) uses the four domains SC, CHS, T and WI, whereas the answer to question (10) uses only the two domains T and WI.

A lot of the questions also require the same domains to be answered. However, the developers expressed a desire for the answer to be presented in different ways. “Who is working on what?”(1) and “What classes has my team been working on?”(16), both require information on SC, CHS, T, and WI to lead to a meaningful answer. However, based on developers’ statements, “Who is working on what?” should display the team members first and the work they have done below with the changed source code last, whereas “What classes has my team been working on?”, the developer wanted to see the elements in the opposite order. In Table 5.1, (1)-(8) and (13)-(22) are two blocks of such questions, where the questions require the same domains but different presentations.

A last subtlety comes with the selection of the actual information. Similar questions often just differ in small details of the information of relevance. However, this difference influences the size of the result and the ease of interpreting it. For example, “What has changed between two builds and who has changed it?”(24) as well as “Who has made changes to my classes?”(25) require information from the same domains. The latter question is however only looking for classes the developer is working on, whereas the first (24) refers to all classes of the project.

5.1.3 Threats

Because of difficulties we had getting subjects to articulate questions involving multiple domains, in this study, we stimulated developers to think about such questions by demonstrating a very early version of the tool we subjected to later testing (Section 6.2). This demonstration may have biased developers to state questions answerable with our approach. We believe this threat to validity is small as other researchers have found similar questions (e.g., [55–57]), adding credence to these being questions developers ask when working. The list of questions we present is partial and is not representative of all multi-domain questions a developer may ask.

5.2 Answering Questions Using Existing Approaches

To answer questions over multiple kinds of software system information, there are mainly two existing classes of approaches: linked views in a programming environment and query languages. We consider whether and how each of these broad classes of approaches supports the answering of the 78 questions we identified from developers.

5.2.1 Using an Integrated Development Environment

Current integrated development environments (IDEs) provide one or more views for each kind of information. Answering the 78 questions in these environments requires a developer to follow links between related information across multiple views and requires a developer to manually correlate the viewed information. Rational Team Concert (RTC), a team collaboration platform on top of the Eclipse IDE, should be well-positioned to answer questions across multiple domains of information as it provides support for a broader range of information than other IDEs. To see how a developer might use an IDE to answer a question of interest, let us consider the use of RTC to answer two of the 78 questions questions adapted to data from a commercial software system development. We provide more detail on these questions and the data later in Section 6.3.2.

The first question we consider is “In the last week, who on the “...” team has been working on the classes in package “...”?” (Q3 in Table 6.3). To answer this question in RTC, one would start out from the package explorer, a view that

presents a hierarchy of Java code (projects, packages, classes and more). To investigate who made changes to the package, the developer must click through the hierarchy to reach the class level and then must open the change history for each class in the history view by selecting the corresponding action within the context menu of each class. Once the change information is populated in the history view, the developer must identify the changes that were made in the last week by looking at the column on the creation date and then he has to determine who made the changes from the column on the creator. He has to do this—opening the history view and identifying the relevant information in it—for each class in the package under consideration. For the commercial project data we use later in Section 6.3, a developer has to open the history view for 83 different classes and identify the team members that made changes over the last week. Switching back and forth 83 times between the two views, the package explorer and the history view, and scanning the information within the view is tedious and time consuming. For the data we considered, all of this work results in determining only two developers who made changes to classes in the package over the last seven days.

The second question we consider is “What is the most popular class in the SCM project? (Q6 in Table 6.3)”. An answer to this question requires determining which class has changed most over the last month. As the views in RTC either focus on source code or on change sets, answering this question in RTC is infeasible for a human to perform. Starting from the source, for the commercial project data we consider, a developer would have to go through each of the 200 packages of the project, open the history view for each of the more than 1000 classes in those packages and count the number of entries in the view. From the change explorer which shows change set information, the question cannot be answered as the change explorer does not provide the number of changes that were made to each class.

5.2.2 Using a Query Language

Query languages, such as SQL, are sufficiently expressive to answer questions over multiple kinds of information. However, the use of a query language is not trivial; a developer has to be trained to use the query language and even once trained,

writing the queries is not always an easy task.

Assuming that different kinds of information are stored in different tables, answering a question such as “In the last week, who on the SCM team has been working on the classes in package A?” (Q3 in Table 6.3) in SQL can be achieved by joining information on the team, the change sets and source code. The following query provides an example of how this question might be answered.

```
SELECT t.name
FROM teamInfo AS t, changeSet AS chs,
     sourceCode AS sc
WHERE t.groupId = 'SCM' AND
     chs.resolved >= '#7_DAYS_AGO' AND
     sc.packageId = 'A' AND
     t.name = chs.author AND
     chs.changedElement = sc.id
ORDER BY t.name
```

Although this query looks simple, it is not easy to get it right on the first try. A study on query languages has shown that users with some computer and little database experience, after substantial training (1.5 hours), can write queries that join elements from two domains in a mean time of 5.1 minutes [15]. Considering that the 78 questions we identified require on average 2.9 information fragments for the answer, a developer would, on average, likely spend over 5 minutes to answer a single question. Given the vast amount of questions a developer faces, he would have to continuously write or adapt queries. We believe that this approach is not practical.

5.3 Information Fragment Model

To enable a developer to practically answer the wide range of questions that may arise, we introduce a model that supports the composition and presentation of information fragments.

5.3.1 Example of Use

We introduce our model by showing how it can be used to answer a specific version of the more generic question, “What have people been working on?” (Question 10 in Table 5.1). We define the approach in Section 5.3.2 to Section 5.3.4.

Consider a software developer Sue who gets back to work after a week of holidays. When she starts working, she wants to know what other developers have been working on while she was away. For this question, Sue is interested in two different information domains: teams and work items. From these two domains, Sue is interested in certain portions of information, which we refer to as *information fragments*. One information fragment consists of the developers of her team (Figure 5.1(a)), and the second consists of the work items on which people have been working (Figure 5.1(b)). In our approach, information fragments are modeled as graphs with nodes and edges. Nodes represent uniquely identifiable items with properties; at least one property is specified as a unique identifier (e.g., *id* in Figure 5.1(b)). For example, a work item includes an identifier (*id*), a creator and an owner property. Edges represent relationships between items, such as a “duplicate-of” relationship between two work items, stating that a work item is a duplicate of another work item.

Each fragment in isolation is not meaningful to Sue. However, by composing these two fragments, Sue can create the context that allows her to answer the question at hand. Our approach provides *composition operators* to compose information fragments. When Sue composes the two information fragments using \otimes_{id} , a new information fragment (Figure 5.1(c)) is created from the input fragments with new edges introduced between nodes based on the nodes’ properties. For example, in the new information fragment, a new “owner” edge is created between David and the defect 303, because the owner property of the node representing defect 303 matches the identifier of the node representing David. Another edge is created between Julie and work item 316 because Julie is the owner and creator of 316.

As the answer to her question, Sue likes to see the work items ordered by developers. Therefore, she chooses a presentation that orders nodes of the team fragment (*t*) above the ones from the work item fragment (*wi*); this presentation is referred to as a projection ($\phi(t, wi)$). Figure 5.1(d) shows the result of this projection. In

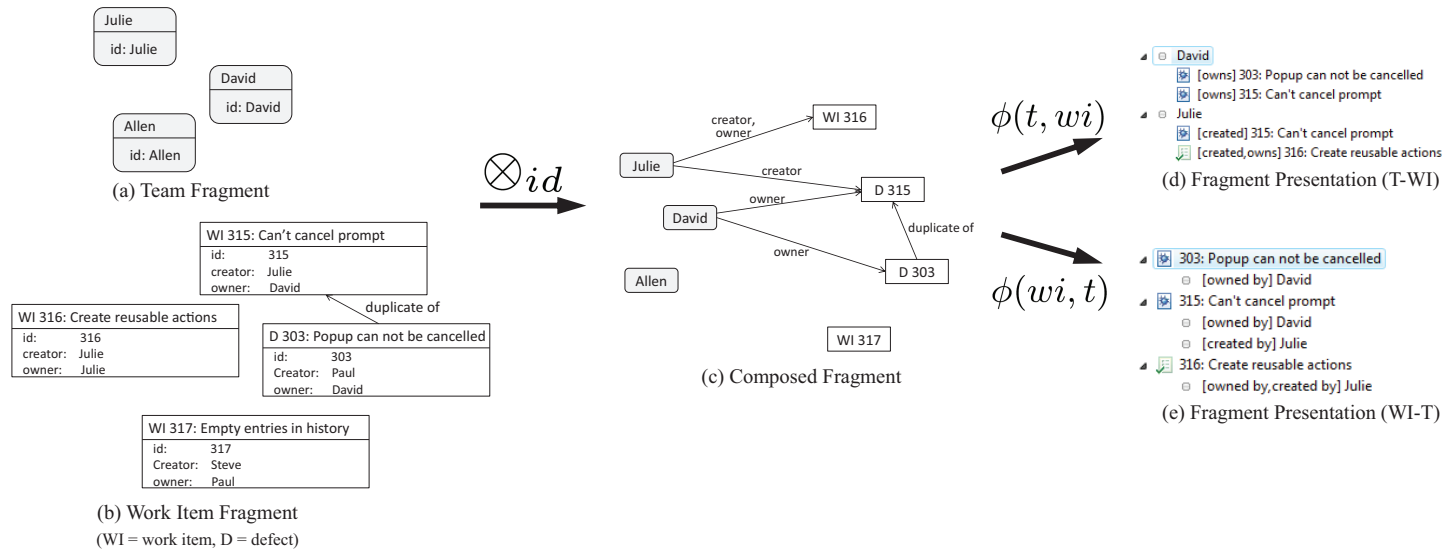


Figure 5.1: Approach to Answer the Question “What have people been working on?”

this presentation, her teammate Allen does not show up in the final presentations as he did not work on, and is thus not connected to, any work item in the fragment.

Alternatively, Sue might be interested in seeing her team members in context of the work items on which they have been working. In this case, she would use a projection $\phi(wi, t)$. This projection shows the work items her fellow team members have been working on first and the developers below (Figure 5.1(e)). By separating the presentation from the composition of the information, either interpretation of the question is easily supported.

5.3.2 Information Fragments

An information fragment is a portion of development information recorded about the software system of interest. We describe an information fragment, building from the foundations of a fragment: nodes and edges.

A node in an information fragment captures an item of information, such as defect 303 or team member Julie. Each node must be uniquely identifiable within its *domain* of information, where a domain defines the set of *types* for nodes in that domain, e.g., in the source code domain, the type of a node can be field, method or class. Each node has, based on its type, a set of properties that describe the node. A property is a triple of name, value and domain that denotes the domain of information to which the property refers, e.g., the owner property of work item 315 in Figure 5.1 has name “owner”, value “David” and refers to domain “teams”. Each node also has an *id* property that uniquely identifies the node and refers to the domain of the node. An exemplary list of domains, types and properties, is given in Table 5.2; for instance, a defect of the work items domain has an identifier and information on the creator and owner of the defect, amongst other properties. Although all types shown in Table 5.2 have the same properties within a domain, in general, this need not be the case.

An edge represents a relationship between nodes. An edge is directed and can be labeled with an arbitrary labeling function. The intuition behind an edge is that it can be based on explicit information, such as a method call, or can be implicit, inferred from the nodes, such as a relationship between a work item and its author (team member).

Table 5.2: Sample Node Domains, Types & Properties

Domain	Types	Properties
<i>source code (SC)</i>	class, method, field	id, referenced source code elements
<i>work items (WI)</i>	defect, work item, plan item	id, creator, owner, linked change sets, linked comments
<i>change sets (CHS)</i>	change set	id, author, changed source code elements
<i>teams (T)</i>	team, team member	id, name
<i>comments (CO)</i>	comment	id, text, author
<i>web/wiki (WW)</i>	web page	id, text, last updated

An information fragment is a graph $F = (V, E)$, with a set of nodes V and a set of edges E . An information fragment can either be a base information fragment (hereafter referred to as a base fragment) or a composition of base fragments. In a base fragment, all nodes are from a single domain. A node can be part of several base fragments. Base fragments can be composed to new information fragments using composition operators as described in the next section.

5.3.3 Composition Operators

A composition operator $\otimes : F_1 \times F_2 \mapsto F'$ takes two information fragments $F_1 = (V_1, E_1)$ and $F_2 = (V_2, E_2)$ and creates a new information fragment $F' = (V', E')$. The set of nodes of the new information fragment is the union of nodes ($V' = V_1 \cup V_2$). E' ($E' = E_1 \cup E_2 \cup E^*$) is the union of E_1 and E_2 and E^* ($E^* \subseteq E_1 \times E_2$), a set of newly created directed, labeled edges that are defined based on the specific composition operator being used. Furthermore, we restrict the composition operators to be commutative. Thus, the order of the input fragments does not matter. This constraint is important for the separation of composition and presentation.

We have determined that only two composition operators are needed to support answering the questions empirically determined from professional programmers detailed in Table 5.1: an identifier matching and a text based matching composition operator.

ID Matching

The id composition operator, denoted by \otimes_{id} , creates a new edge when the identifier property (id) of one node n_1 exactly matches the value of a property p_2 of another node and the domain of property p_2 matches the domain of the id of node n_1 . For example, when applying \otimes_{id} to the team and the work item fragment in Figure 5.1, a new edge is created between the node Julie from the teams domain and work item 315, since the id property of Julie matches the value of the creator property of work item 315 and the creator property is of domain teams. The label of the newly created edge is the name of property p_2 . In our example, the newly created edge between the team member and the defect is labeled *creator*.

Text Matching

The text matching operator, denoted by \otimes_t , creates new edges when there is an exact textual match between the identifier property of one node with a textual property in another node. This operator can be generalized by extending it with a similarity measure. The similarity of the identifier on the one side and the textual match on the other side can be based on a text matching measure, such as the Levenshtein [36] or the Hamming distance [40], with a threshold for similarity used to determine if there is a match. Consider a team member and a comment node. If the identifier property of the team member has a textual match in the text property of the comment with a high enough similarity, a new edge will be created. This new edge between the team member and the comment has properties that describe the match, such as the similarity value and the location of the match in the comment. We discuss issues and possibilities surrounding text matching later in the thesis (Section 7.2.2).

5.3.4 Presentation

Our interviews (Section 5.1) identified variations in the ways developers would like a composed information fragment to be presented. To support this variety, our approach provides a *projection* function that transforms an information fragment (a graph) into a set of trees. Specifically, given an information fragment $F = (V, E)$ and an ordering (bf_1, \dots, bf_n) of the base fragments the information fragment is

composed of, a projection denoted by $\phi_{(bf_1, \dots, bf_n)}$ creates a set of trees, TS . All tree roots are nodes of base fragment bf_1 for which a path to nodes of base fragment bf_n exists, so that the nodes on the path follow the given order (bf_1, \dots, bf_n) . For convenience, we introduce a mapping function l_F ($l_F : v \mapsto \{F_1, \dots, F_n\}$ with $F_i = (V_i, E_i)$ and $v \in V_i$ for $i = 1, \dots, n$) that maps each node to the set of base fragments of which the node is a part. Formally, for each path (v_1, \dots, v_n) through a created tree $T = (V_T, E_T)$ with $T \in TS$, $V_T \subseteq V$ and $E_T \subseteq E$ the following holds: $bf_j \in l_F(v_j)$ for $j = 1, \dots, n$ and there is no other tree $T^* \in TS$ that contains the path. In our example, $\phi_{(t, wi)}$ creates two trees (shown in Figure 5.1(d)) that represent all possible paths of length two from a node of the team fragment to a node of the work item fragment.

Sometimes, a developer needs to count nodes on a particular level in the presentation for a summary. For instance, several questions are about the relative occurrences of composed information, such as “What’s the most popular class” (Question 38 in Table 5.1). Therefore, our model provides a *counting* function, denoted by $\sigma_{(levelOf(bf))}$, that counts all nodes on the level of a base fragment bf in the set of trees. For example, to find out how many work items each team member worked on in our example, a developer can count the nodes of the team fragment level ($\sigma_{(levelOf(t))}$) in Figure 5.1(e). This tells him that Julie and David each occur twice in the trees and thus worked on two work items. For simplicity, we use the short form, $\sigma_{(bf)}$, to denote the counting function.

Chapter 6

Evaluation of the Information Fragments Model

To investigate whether the information fragments model can support a developer in answering questions that span across multiple kinds of information according to the developer’s personal preferences, we examined two aspects: the model’s expressibility and the model’s usefulness to developers.

We present how all 78 questions determined in the interview study *Study_{EXP_FR}* can be expressed using our model (Section 6.1). To allow the investigation of the usefulness of the information fragments model, we implemented a prototype supporting the model within an integrated development environment (Section 6.2). We present the results of a case study, *Study_{INFR}*, with 18 professional developers, showing that the developers were able to easily apply the model to successfully answer 94% of eight questions posed (Section 6.3).

6.1 Applying the Model

We have applied our model to answer the 78 questions presented in Section 5.1. Table 5.1 shows the domains and composition operators needed to answer each question as it was intended by the developers who stated the question. The compositions shown in Table 5.1 rely on 22 information fragments presented in Table 6.1. These fragments use various abstractions that we believe are faithful to the mean-

ing of the developers' questions. For instance, we grouped developers' statements such as "newly resolved" and "lately" to "recently". We use the phrase "of interest" for fragments that refer to a small and specific set of nodes in which a developer was interested. The major difference between fragments is how the information source is filtered by time.

Based on developers' comments about what they wanted to see for the answer, Table 6.2 shows the answer to each of the 78 questions. We present the answers to 9 of the 78 questions in more detail. We chose the following 9 questions to cover all eight domains of information, as well as text and id matching, projection and counting. The selected questions also involve multiple information fragments from the same domain.

What have people been working on? (10)

As an answer, the developer who stated this question wanted to see the list of his team members with work items on which they have been active beneath the relevant team member. Applying our model, we first need access to the information fragments of interest:

T_1 the direct team members of the developer,

WI_3 work items resolved recently and in progress.

We then apply the composition operator that matches identifiers (\otimes_{id}). The composition of these fragments can be expressed as

$$T_1 \otimes_{id} WI_3.$$

As the composition is commutative, the order in which the composition takes place does not matter.

To display the composed information as desired by the developer we apply a suitable projection (i.e., team first and then work items)

$$\phi_{(T_1, WI_3)}.$$

Overall, the answer to the question is

$$\phi_{(T_1, WI_3)}(T_1 \otimes_{id} WI_3).$$

Who is working on what? (1)

This question is similar to the one above, but the developer who stated it, wanted

Table 6.1: Information Fragments For Answering Developer's Questions

T_1	direct team members of the developer
T_2	all team members on the project
T_3	team members of interest
T_4	the developer
WI_1	work items resolved recently
WI_2	work items in progress
WI_3	work items resolved recently and in progress
WI_4	work items of interest
WI_5	all work items for last couple of months
CHS_1	change sets recently delivered
CHS_2	change sets in progress
CHS_3	change sets recently delivered and in progress
CHS_4	change sets of interest
CHS_5	all change sets for last couple of months
SC_1	source code of the project
SC_2	source code of interest
CO_1	comments recently created
CO_2	all comments for last couple of months
ST_1	stack trace of interest
TC_1	test cases of project
TC_2	test cases of interest
W_1	web site(s) of interest

to also see the changes made to the code. The additional information fragments are:

CHS_3 change sets recently delivered and in progress,

SC_1 source code of the project.

To reach the answer with our model, we use:

$$\phi_{(T_1, WI_3, CHS_3, SC_1)}(SC_1 \otimes_{id} CHS_3 \otimes_{id} WI_3 \otimes_{id} T_1).$$

What classes has my team been working on? (16)

Answering this question requires the same information fragments as question (1).

However, the developer's focus for this question was on the code and he wanted to

see the work items and team members in the context of his current project. Compared to question (1), the answer differs only in the order of the projection:

$$\phi_{(SC_1, CHS_3, WI_3, T_1)}(SC_1 \otimes_{id} CHS_3 \otimes_{id} WI_3 \otimes_{id} T_1).$$

Who owns/modified this piece of code most? (29)

Answering this question requires three information fragments:

- T_2 all team members on the project,
- CHS_5 all change sets for the last couple of months, and
- SC_2 source code of interest.

To find out who made most changes over the last couple of months, the composed information is projected and then the occurrences of the team members are counted:

$$\sigma_{(T_2)}(\phi_{(SC_2, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_2)).$$

Which conversations in work items have I been mentioned [in]? (77)

The developer wanted to see the comments he is mentioned in ordered by work items.

- T_4 the developer,
- CO_1 comments recently created,
- WI_4 work items of interest.

As textual matches of the developer in the comments are of interest, the team fragment is composed using the text matching composition operator (\otimes_t). Overall, with work items first and comments second, this results in:

$$\phi_{(WI_4, CO_1, T_4)}(T_4 \otimes_t CO_1 \otimes_{id} WI_4).$$

Who is using that API [that I am about to change]? (26)

To determine which other code is using the API the developer is about to change, two information fragments of source code are required: the API (SC_2) and the code of the project (SC_1). With the *referenced elements* property of source code nodes, the \otimes_{id} composition operator will create links between the API nodes and other source code elements that reference (use) them. To determine which developers use the API, we now only have to compose the change set information (CHS_5) and an information fragment of all team members (T_2):

$$\phi_{(SC_2, SC_1, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_1 \otimes_{id} SC_2).$$

What's the most popular class? [Which class has been changed most?] (38)

Answering this question requires two information fragments: SC_1 and CHS_5 . To find out, which class has the most change sets delivered to it, the projected information is projected and then the classes are counted:

$$\sigma_{(SC_1)}(\phi_{(CHS_5, SC_1)}(CHS_5 \otimes_{id} SC_1)).$$

Which changes caused the tests to fail and thus the build to break? (63)

The developer wanted to trace starting from the stack trace (ST_1), the test cases that failed (TC_1) and the corresponding source code elements (SC_1). From there, he wanted to see the actual code that is executed by the test cases (another occurrence of SC_1) and then find the changes (CHS_1) that were recently made to the source code. Overall, he was interested in the changes that caused the test cases to fail, so together with the projection with the change sets first, the answer to the question is achieved by:

$$\phi_{(CHS_1, SC_1, SC_1, TC_1, ST_1)}(ST_1 \otimes_{id} TC_1 \otimes_{id} SC_1 \otimes_{id} SC_1 \otimes_{id} CHS_1).$$

Which API has changed (check on web site)? (68)

To stay aware of changes to a third-party API used, a developer stated that he often monitors web sites that announce API changes. Using the \otimes_t composition operator, the textual matches of the usage of the API in the source code and the API change announcements on the monitored web site can be determined. Projecting the composed fragment to source code first shows the developer the elements that call API that has a match on the web site:

$$\phi_{(SC_1, W_1)}(SC_1 \otimes_t W_1).$$

Table 6.2: Using the Model to Answer all 78 Questions.

1. Who is working on what?	$\phi_{(T_1, WI_3, CHS_3, SC_1)}(T_1 \otimes_{id} WI_3 \otimes_{id} CHS_3 \otimes_{id} SC_1)$
2. What are they [coworkers] working on right now?	$\phi_{(T_1, WI_2, CHS_2, SC_1)}(T_1 \otimes_{id} WI_2 \otimes_{id} CHS_2 \otimes_{id} SC_1)$
3. What have other people been working on?	$\phi_{(T_3, WI_1, CHS_1, SC_1)}(T_3 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
4. How much work [have] people done?	$\phi_{(T_1, WI_1, CHS_1, SC_1)}(T_1 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
5. Who changed this [code], focused on person?	$\phi_{(T_2, WI_1, CHS_1, SC_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
6. Who to assign a code review to? / Who has the knowledge to do the code review?	$\phi_{(T_1, WI_1, CHS_1, SC_2)}(T_1 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
7. What [have] people done lately?	$\phi_{(T_1, WI_1, CHS_1, SC_1)}(T_1 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
8. Who is working on what at the moment?	$\phi_{(T_1, WI_2, CHS_2, SC_1)}(T_1 \otimes_{id} WI_2 \otimes_{id} CHS_2 \otimes_{id} SC_1)$
9. What has [a particular team member] been doing?	$\phi_{(T_3, CHS_1, SC_1)}(T_3 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
10. What have people been working on?	$\phi_{(T_1, WI_3)}(T_1 \otimes_{id} WI_3)$
11. Which code reviews have been assigned to which person?	$\phi_{(T_1, WI_4)}(T_1 \otimes_{id} WI_4)$
12. Who to assign a code review to? / Who has time for a code review?	$\phi_{(T_1, WI_2)}(T_1 \otimes_{id} WI_2)$
13. What is the evolution of the code?	$\phi_{(SC_2, CHS_5, WI_5, T_2)}(T_2 \otimes_{id} WI_5 \otimes_{id} CHS_5 \otimes_{id} SC_2)$
14. Why were they [these changes] introduced?	$\phi_{(SC_2, CHS_4, WI_1, T_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_4 \otimes_{id} SC_2)$
15. Who made a particular change and why?	$\phi_{(SC_2, CHS_4, WI_1, T_1)}(T_1 \otimes_{id} WI_1 \otimes_{id} CHS_4 \otimes_{id} SC_2)$

Table6.2: (Continued)

-
16. What classes has my team been working on?
 $\phi_{(SC_1, CHS_3, WI_3, T_1)}(T_1 \otimes_{id} WI_3 \otimes_{id} CHS_3 \otimes_{id} SC_1)$
17. What are the changes on newly resolved work items related to me?
 $\phi_{(SC_1, CHS_1, WI_1, T_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
18. Who is working on the same classes as I am and for which work item?
 $\phi_{(SC_2, CHS_2, WI_2, T_2)}(T_2 \otimes_{id} WI_2 \otimes_{id} CHS_2 \otimes_{id} SC_2)$
19. Who changed this [code], focused on code?
 $\phi_{(SC_1, CHS_1, WI_1, T_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
20. What is the whole history of this file?
 $\phi_{(SC_2, CHS_5, WI_5, T_2)}(T_2 \otimes_{id} WI_5 \otimes_{id} CHS_5 \otimes_{id} SC_2)$
21. What has been happening on [this] class?
 $\phi_{(SC_2, CHS_1, WI_1, T_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
22. What [have] people changed lately?
 $\phi_{(SC_1, CHS_1, WI_1, T_2)}(T_2 \otimes_{id} WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
23. What changes have been made and why?
 $\phi_{(SC_1, CHS_1, WI_1)}(WI_1 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
24. What has changed between two builds [and] who has changed it?
 $\phi_{(SC_1, CHS_4, T_2)}(T_2 \otimes_{id} CHS_4 \otimes_{id} SC_1)$
25. Who has made changes to my classes?
 $\phi_{(SC_2, CHS_1, T_2)}(T_2 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
26. Who is using that API [that I am about to change]?
 $\phi_{(SC_2, SC_1, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_1 \otimes_{id} SC_2)$
27. Who created the API [that I am about to change]?
 $\phi_{(SC_2, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_2)$
28. Who owns this piece of code? / Who modified it the latest?
 $\phi_{(SC_2, CHS_1, T_2)}(T_2 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
29. Who owns this piece of code? / Who modified it most?
 $\sigma_{(T_2)}(\phi_{(SC_2, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_2))$
30. Who to talk to if you have to work with packages you haven't worked with?
 $\sigma_{(T_2)}(\phi_{(SC_2, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_2))$
-

Table6.2: (Continued)

31. How much has changed [in the project code]?	$\phi_{(SC_1, CHS_1, T_2)}(T_2 \otimes_{id} CHS_1 \otimes_{id} SC_1)$
32. [Is anyone] intending to commit anything to that class?	$\phi_{(SC_1, CHS_2, T_2)}(T_2 \otimes_{id} CHS_2 \otimes_{id} SC_1)$
33. Where have changes been made related to you?	$\phi_{(SC_2, CHS_1, T_2)}(T_2 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
34. Who is responsible for this code? (Who made the latest change?)	$\phi_{(SC_2, CHS_1, T_2)}(T_2 \otimes_{id} CHS_1 \otimes_{id} SC_2)$
35. Which team is responsible for this code? (Who has made most changes to the code?)	$\sigma_{(T_2)}(\phi_{(SC_2, CHS_5, T_2)}(T_2 \otimes_{id} CHS_5 \otimes_{id} SC_2))$
36. What classes have been changed?	$\phi_{(SC_1, CHS_1)}(CHS_1 \otimes_{id} SC_1)$
37. [Which] API has changed (to see which methods are not supported any more)?	$\phi_{(SC_2, CHS_1)}(CHS_1 \otimes_{id} SC_2)$
38. What's the most popular class? [Which class has been changed most?]	$\sigma_{(SC_1)}(\phi_{(CHS_5, SC_1)}(CHS_5 \otimes_{id} SC_1))$
39. Which other code that I worked on uses this code pattern / utility function?	$\phi_{(SC_1, SC_1, CHS_4)}(CHS_4 \otimes_{id} SC_1 \otimes_{id} SC_1)$
40. Which code has recently changed that is related to me?	$\phi_{(SC_2, SC_1, CHS_1)}(CHS_1 \otimes_{id} SC_1 \otimes_{id} SC_2)$
41. How do recently delivered changes affect changes that I am working on?	$\phi_{(CHS_4, SC_1, CHS_1)}(CHS_4 \otimes_{id} SC_1 \otimes_{id} CHS_1)$
42. What code is related to a change?	$\phi_{(CHS_4, SC_1, SC_1)}(CHS_4 \otimes_{id} SC_1 \otimes_{id} SC_1)$
43. Where has code been changing [this week]?	$\phi_{(SC_1, CHS_4)}(CHS_4 \otimes_{id} SC_1)$
44. Which classes have been changed between two builds?	$\phi_{(SC_1, CHS_4)}(CHS_4 \otimes_{id} SC_1)$
45. What is going on [in a package]?	$\phi_{(SC_2, CHS_1)}(CHS_1 \otimes_{id} SC_2)$

Table6.2: (Continued)

46. [Which] changes [have been made] between these days or after this day?	$\phi_{(SC_1, CHS_4)}(CHS_4 \otimes_{id} SC_1)$
47. What classes in this component were modified since version [...]?	$\phi_{(SC_2, CHS_4)}(CHS_4 \otimes_{id} SC_2)$
48. What is the recent activity on a plan item?	$\phi_{(WI_4, WI_3, CO_1)}(WI_4 \otimes_{id} WI_3 \otimes_{id} CO_1)$
49. Which features and functions have been changing?	$\phi_{(WI_4, CO_1)}(WI_4 \otimes_{id} CO_1)$
50. Has progress been made on blockers (blocking work items) in your milestone?	$\phi_{(WI_4, CO_1)}(WI_4 \otimes_{id} CO_1)$
51. Which work items/plan items are most active?	$\sigma_{(WI_5)}(\phi_{(CO_1, WI_5)}(WI_5 \otimes_{id} CO_1))$
52. How active is the [plan item]? [How many comments were made on related work items?]	$\phi_{(WI_4, WI_3, CO_1)}(WI_4 \otimes_{id} WI_3 \otimes_{id} CO_1)$
53. Are there any new comments on interesting work items?	$\phi_{(WI_4, CO_1)}(WI_4 \otimes_{id} CO_1)$
54. [What are the] emails related to line items and defects that are features?	$\phi_{(WI_4, CO_1)}(WI_4 \otimes_t CO_1)$
55. Which work item has recently changed that is related to me?	$\phi_{(T_4, WI_1, CO_1)}(T_4 \otimes_{id} WI_1 \otimes_{id} CO_1)$
56. What are the comments on newly resolved work items that are related to me?	$\phi_{(T_4, WI_1, CO_1)}(T_4 \otimes_{id} WI_1 \otimes_{id} CO_1)$
57. Is progress (changes) being made on plan items?	$\phi_{(WI_4, CHS_3)}(WI_4 \otimes_{id} CHS_3)$
58. What is the activity on a line item (feature)?	$\phi_{(WI_4, CHS_3)}(WI_4 \otimes_{id} CHS_3)$
59. What caused this build to break? (Which change caused the stack trace?)	$\phi_{(ST_1, SC_1, CHS_1)}(ST_1 \otimes_{id} SC_1 \otimes_{id} CHS_1)$

Table6.2: (Continued)

60. What has caused this build to break? (look at stack trace and intersect with change sets)	$\phi_{(ST_1, CHS_1)}(ST_1 \otimes_{id} CHS_1)$
61. Who caused this build to break? (Who owns the broken tests?)	$\sigma_{(T_2)}(\phi_{(TC_2, SC_1, CHS_1, T_2)}(TC_2 \otimes_{id} SC_1 \otimes_{id} CHS_1 \otimes_{id} T_2))$
62. Who changed the test case most recently that caused the build to fail?	$\phi_{(T_2, CHS_1, SC_1, TC_2)}(TC_2 \otimes_{id} SC_1 \otimes_{id} CHS_1 \otimes_{id} T_2)$
63. Which changes caused the tests to fail and thus the build to break?	$\phi_{(CHS_1, SC_1, SC_1, TC_1, ST_1)}(ST_1 \otimes_{id} TC_1 \otimes_{id} SC_1 \otimes_{id} SC_1 \otimes_{id} CHS_1)$
64. Who owns a test case? (Who resolved the last work item that fixed the test case?)	$\phi_{(TC_2, SC_1, CHS_1, WI_1, T_2)}(TC_2 \otimes_{id} SC_1 \otimes_{id} CHS_1 \otimes_{id} WI_1 \otimes_{id} T_2)$
65. Who is responsible for a failing test case? (stack trace)	$\phi_{(T_1, CHS_1, SC_1, TC_1, ST_1)}(ST_1 \otimes_{id} TC_2 \otimes_{id} SC_1 \otimes_{id} CHS_1 \otimes_{id} T_2)$
66. How do test cases relate to work items?	$\phi_{(TC_1, SC_1, SC_1, CHS_1, WI_1)}(TC_1 \otimes_{id} SC_1 \otimes_{id} SC_1 \otimes_{id} CHS_1 \otimes_{id} WI_1)$
67. How do test cases relate to packages/classes?	$\phi_{(TC_1, SC_1, SC_1)}(TC_1 \otimes_{id} SC_1 \otimes_{id} SC_1)$
68. Which API has changed (check on web site)?	$\phi_{(SC_1, W_1)}(SC_1 \otimes_t W_1)$
69. [Is an entry] in newsgroup forum addressed to me because of the class mentioned?	$\phi_{(SC_2, W_1)}(SC_2 \otimes_t W_1)$
70. What is coming up next week [for my team]? [What is my team doing?]	$\phi_{(T_1, W_1)}(T_1 \otimes_t W_1)$
71. What am I supposed to work on [plan on wiki]?	$\phi_{(W_1, T_4)}(T_4 \otimes_t W_1)$
72. Who has to do what? [team activity]	$\phi_{(T_1, W_1)}(T_1 \otimes_t W_1)$
73. How is the team organized?	$\phi_{(T_1, CHS_5, SC_1)}(T_1 \otimes_{id} CHS_5 \otimes_{id} SC_1)$

Table6.2: (Continued)

74. Who has made changes to [a] defect?	$\phi_{(WI_4, CHS_1, T_2)}(WI_4 \otimes_{id} CHS_1 \otimes_{id} T_2)$
75. Who has made comments in defect?	$\phi_{(WI_4, CO_1, T_2)}(WI_4 \otimes_{id} CO_1 \otimes_{id} T_2)$
76. [What is] the collaboration tree around a feature?	$\phi_{(WI_4, CO_2, T_2)}(WI_4 \otimes_{id} CO_2 \otimes_{id} T_2)$
77. Which conversations in work items have I been mentioned [in]?	$\phi_{(WI_4, CO_1, T_4)}(WI_4 \otimes_{id} CO_1 \otimes_t T_4)$
78. What are people commenting [on] all work items I am involved with?	$\phi_{(T_4, WI_3, CO_1)}(WI_3 \otimes_{id} CO_1 \otimes_{id} T_4)$

6.2 Prototype

We have implemented a prototype that supports our information fragments model. Our prototype extends IBM Rational Team Concert (RTC)¹, a team collaboration platform on top of the Eclipse IDE².

6.2.1 Information Fragments

Our prototype supports five domains of information: work items, source code (Java packages, classes, methods and fields), change sets, teams (teams and team members) and test cases (see Section 7.2.5 for the latter). We enhanced existing views for these domains in RTC to support the creation of information fragments from elements selected in the view. The next phase of our research will focus on better support for selecting fragments (see Section 7.2.4).

6.2.2 Composition

With the prototype, a developer can compose information fragments by adding them to the *composed viewer*. Figure 1.1 presented in Section 1.2 shows the answer to “Who is working on what?” (Question 1 in Table 5.1) in the composed viewer. The answer was created by adding the team, work item, change set and source code fragments described in Section 6.1 to the viewer. The appropriate composition operator is applied automatically when information is placed into the composed viewer. In this example, the relations that can be seen in the view are all based on the identifier matching operator.

Figure 6.1a displays the answer to the question “Does a work item mention a class of interest to me?” (similar to question 69 in Table 5.1) in the composed viewer. The answer was created by adding two fragments, the source code of interest and the newly created work items, to the viewer. In this case, the result contains edges (named “matches”), created automatically by the text matching operator, based on when the class or method name matched text to the work item summary. In contrast to the identifier matching operator that creates edges only for exact matches, the text matching operator can account for variations in the tex-

¹jazz.net, verified 11/11/10

²www.eclipse.org, verified 11/11/10

tual match, such as word synonyms or even spelling mistakes. This approximation may result in information being presented that is not relevant to the developer's question. We leave the determination of appropriate text matching operators and representation of variance in edge confidence to future research.

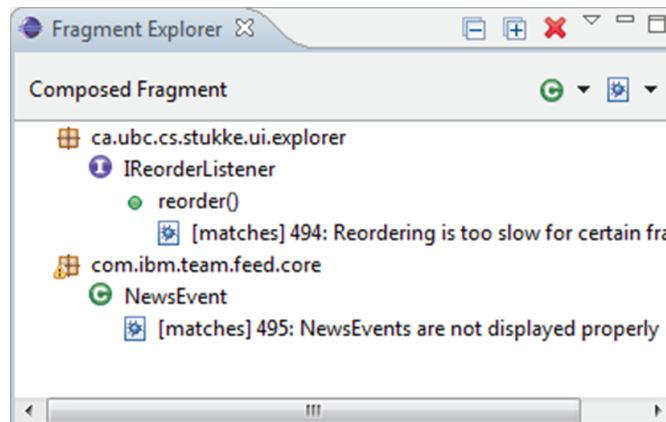
We discuss how to determine which composition operators to apply in Section 7.2.1.

6.2.3 Presentation

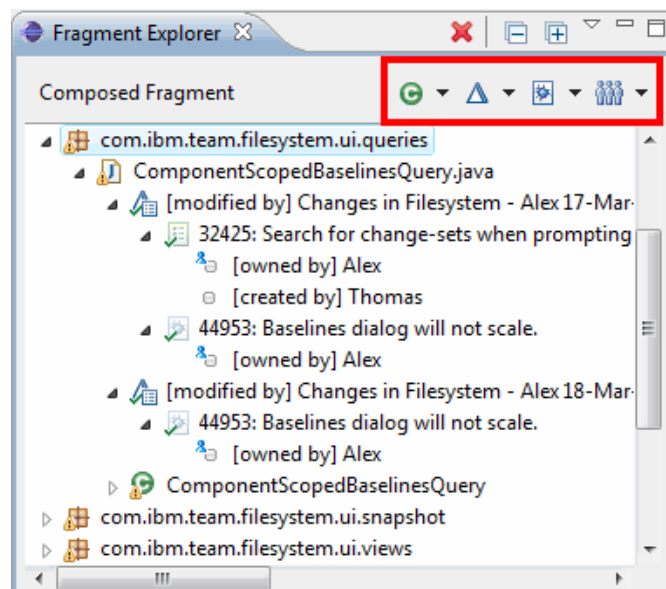
The prototype supports projection and counting of our model as well as a feature for hiding certain information. Figure 1.1 displays the answer to “Who is working on what?” as described in Section 6.1. The projection order of the base fragments is represented by the icons in the *fragment bar* in the upper right corner of the view. If the developer is interested in a code-centric interpretation of the same question, he only has to change the order of the original base fragments by dragging and dropping the icons in the fragment bar into the appropriate order. This reordering action changes the projection; the underlying composed information fragment does not change. The resulting view is shown in Figure 6.1b. The highlighted rectangle in Figure 6.1b shows the fragment bar with the projection order.

Answering a question such as which of the team members made the most changes requires a count. A developer can perform this count using the drop down menu of the teams icon in the fragment bar. The count functionality counts the occurrences of nodes of a particular base fragment, in this case the team fragment, presented in the view. The developer can also count the children of a specific element. For example, he can count the team members that changed a specific Java package by right-clicking on the Java package in the view and selecting the “Count Team Members” action in the context menu.

As an additional feature for eliding information in the view, the prototype supports a hide action that allows the developer to hide nodes of a certain level. This functionality is again accessible through the drop down menus of each icon in the fragment bar.



(a) Using Text Matching to Find Work Items that Mention Classes of Interest



(b) “Who is working on what?”—Code Centric Presentation

Figure 6.1: Views of the Prototype

6.2.4 Completeness of the Prototype

Our current implementation supports the identifier composition, the text matching composition and the presentation operators of our model. With these operators, the prototype supports the composition and presentation of all 78 questions de-

scribed earlier (Section 5.1). As currently implemented, the prototype supports six domains of information: source code, change sets, teams, work items, web feeds and test cases. Thus, the prototype can be used directly to answer 61 of the 78 questions; direct answers for the remaining questions would require support for comments on work items and stack traces. Extending our initial prototype with information on test cases suggests that adding support for these additional domains is straightforward (Section 7.2.5) when the underlying environment provides access to these domains as uniquely identified items.

6.3 Study

For the information fragment model to be useful, a developer must be able to easily apply the model to answer questions of interest about a software development. To determine whether the model is useful, we posed the following four research questions:

- (1) Can developers use the information fragment model to answer questions that we have demonstrated that developers ask?
- (2) Can developers use the model effectively without requiring a detailed understanding of how the model works? (This condition can improve adoption and use.)
- (3) How do developers use the model?
- (4) What do developers think about the approach?

To answer these questions, we conducted a study in which eighteen professional developers used our prototype tool that supports the information fragments model to answer eight questions selected from those described in Table 5.1. Our study setup was an embedded, multiple-case, replication design.

We chose not to perform a comparative study as the only available approach is to follow links in a development environment. As we argued in Section 5.2, this approach is infeasible to use for realistic data sets.

6.3.1 Subjects

We originally recruited twenty-one developers from two different locations of a multi-national company who were working on four different teams. To be eligible to participate in the study, a developer had to use IBM's Rational Team Concert client in his daily work. To solicit participation, we randomly asked people at the two locations. We report on eighteen of the participants: ten from one location (S1-S10) and eight from the other location (T1-T8). The remaining three individuals had difficulty with the experimental situation; for instance, one subject experienced a strong test anxiety and could not focus. The roles of the eighteen developers ranged from junior developer to team lead and also included one student. The professional experience ranged from seven months to twenty-three years. Three of the developers (T6,T7,T8) had never used the source control mechanism referred to in the study. Six of these eighteen developers participated in our interview-based on developer's questions that we conducted six months previous to this study (see Section 5.1).

6.3.2 Study Method

We selected a set of eight questions from the 78 questions derived from the interviews with developers. Each question was made more specific for two reasons: 1) to reduce the range of interpretations of the questions so that we could compare how the participants answered each question and 2) to match the data available in our study setup. For example, we adapted the question

(16) "What classes has my team been working on?"

to

(Q5) Yesterday, on which classes (of the SCM code) have Alex and Allen on the SCM team been working on and why? For each developer name one class and the reason for the change.

These changes to the question reflect the desire of the developer who originally stated question 16 to see the reason for the changes in terms of work items and reflect the specification of the scope of the team, code and timespan of interest. We

narrowed Q5 to ask about one class for each developer to allow us to determine when a participant's given answer was correct. For Q5, we considered an answer correct if the participant mentioned, for each of Alex and Allen, a class which the respective developer changed yesterday and the work item requiring that change.

Table 6.3 shows the eight specific questions; the second column in Table 6.3 maps the study questions to the original questions shown in Table 5.1. We chose these eight to cover the most common domains as can be seen from the corresponding questions and also to cover a range of complexity and number of domains required for answering each question. Four of the questions (Q1, Q4, Q6, Q8) require information from at least two different domains to be composed, three questions (Q2, Q3, Q7) require information from at least three domains and one (Q5) requires information from four different domains to be composed. One of the questions (Q8) also requires two base fragments from the same domain to be composed for its answer.

The version of the prototype used in the study was adapted with a view to show ten predefined base fragments, such as “all SCM source code” or “work items worked on yesterday”. Figure 6.2 and Table 6.4 provide a list of all ten fragments available to each study participant. We predefined these base fragments to focus the study on the core parts of the model: the composition and projection of information fragments. The predefinition of the fragments simplifies the problem by removing the burden of choosing appropriate fragments from the user; this approach also makes the problem harder by prohibiting the user from just choosing the portions of information he or she was interested in. In Section 7.2.4 we further discuss the selection of information fragments.

The data used for these base fragments was prepared from the history of the Rational Team Concert development from a year ago, with developer names changed. The source code fragments used ranged from 22,000 (SC_1) to 36,000 (SC_2) Java elements, including packages, classes, methods and fields. The change set fragments used ranged from more than 100 (CHS_1) to over 1000 (CHS_3) change sets. The work item fragments used ranged from 50 (WI_1) to more than 350 (WI_2) items. Team fragment (T_1) consisted of one team with 9 developers and team fragment (T_2) consisted of 34 developers from four different teams.

Table 6.3: Study Questions (original developer question in brackets)

Q1	(36)	Which classes in the “...” package of the SCM project have been changed in the last week?
Q2	(25)	In the last week, who on the SCM team has changed classes that I am interested in?
Q3	(5)	In the last week, who on the SCM team has been working on the classes in package “...”?
Q4	(10)	In the last week, who on my team (SCM) was working on which work item? For two different developers name one work item each.
Q5	(16)	Yesterday, on which classes (of the SCM code) have Alex and Allen on the SCM team been working on and why? For each developer name one class and the reason for the change.
Q6	(38)	What is the most popular class in the SCM project? (Which class in the SCM project has been changed most over the last month?)
Q7	(35)	Who on my team is responsible for the classes in package “...” of the SCM project? (Who made most changes to the classes over the last month?) Name the responsible developer for each class that was changed.
Q8	(41)	Name one class that you (Alex) worked on yesterday that was affected by changes delivered in the last week. (How do recently delivered changes affect changes that Alex was working on?)

Table 6.4: The Ten Base Fragments Available to Participants

<i>SC</i> ₁	source code of interest to me (my code)
<i>SC</i> ₂	all SCM source code
<i>CHS</i> ₁	change sets delivered yesterday
<i>CHS</i> ₂	change sets delivered in the last week
<i>CHS</i> ₃	change sets delivered in the last month
<i>T</i> ₁	SCM team
<i>T</i> ₂	Jazz team
<i>T</i> ₃	SCM team without you (Alex)
<i>WI</i> ₁	work items worked on yesterday
<i>WI</i> ₂	work items worked on in the last week

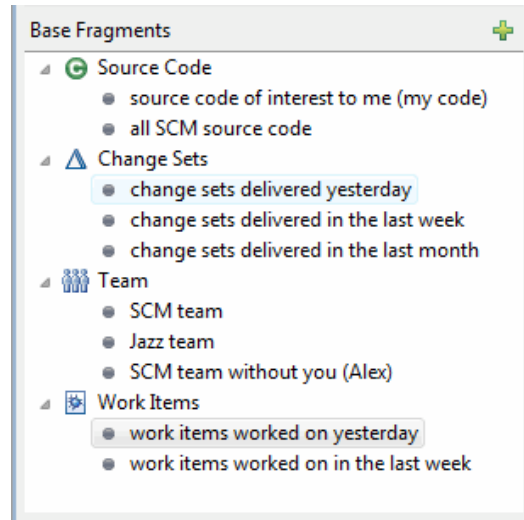


Figure 6.2: Pre-defined Base Fragments in the Prototype

At the start of each study session, a participant worked through a two-page tutorial about the prototype’s features: composition, reordering, counting and hiding. This step took approximately ten to fifteen minutes. The examples used in this tutorial were straightforward, such as relying on the well-known fact that change sets contain links to their authors and, as a result, can be composed with team information. After finishing the tutorial, a participant was given time to read the eight questions (Q1-Q8) and ask clarification questions. A participant had the choice to answer these eight questions in whichever order he preferred, but was told that the questions were ordered from easier to more difficult ones.

Participants were then given time to work on the questions. If a participant spent five minutes on a question and was still not close to an answer we provided a hint. This situation only occurred for Q5 and Q8. For Q5, two participants required a hint about ordering; one participant who had not used the source control system needed a hint about work items containing links to change sets. Six participants for Q8 were given a hint about what the question meant. If a participant was not done with a question after ten minutes we stopped him and asked him to move to another question. We consider ten minutes per question to be a justifiable time limit that does not put too much stress on a participant.

After a participant finished all eight questions, we interviewed the participant about the experience of using the tool.

6.3.3 Data Analysis

For each participant, we captured screen videos as the participant worked and took written notes. From our notes, we determined, for each question, the worst case time that it took the participant to get to the correct answer based on our interpretation of correctness. Only for Q5 was there a distinct difference between our notion of correctness and the participant's notion of correctness. For Q5, eight participants (S3,S6,S7,S10,T3,T4,T6,T8) interpreted the reason for a change as the one-line description of the change set, and did not consider work items. When directed by the experimenter to consider work items, the participants continued working on the question, taking, on average, an extra three minutes to get to the correct answer. We used only these worst case times for our analysis.

We used the video to analyze the interaction of a participant with the features of the tool.

6.3.4 Can Developers Use the Model?

To evaluate our first research question “Can developers use the information fragment model to answer questions that previous developers have posed?”, we consider how many of the eight questions participants were able to answer correctly. If our model is usable, we expect participants to succeed in answering most questions. Moreover, for our model to improve on a query approach, participants should succeed in answering most questions in less than five minutes. As mentioned before (see Section 5.2.2), studies of SQL have shown that users, after substantial training (1.5 hours), use a mean time of 5.1 minutes to write queries that join elements from two domains [15]. Given the, on average, 2.75 domains of information that must be joined to answer any of the eight study questions, a mean time of less than five minutes constitutes a reasonable answering time for these questions.

Table 6.5 shows, for each question in the study, the time and success per developer in each cell, the mean time needed for each question in the last column and the mean time per question for each developer in the bottom-most row. The

Table 6.5: Developer's Results

Question (orig. dev. question)	Time (in minutes) and Success per Question for each Developer																			Mean
	✓ = success, ■ = failure, * = hint given																			
	<i>S1</i>	<i>S2</i>	<i>S3</i>	<i>S4</i>	<i>S5</i>	<i>S6</i>	<i>S7</i>	<i>S8</i>	<i>S9</i>	<i>S10</i>	<i>T1</i>	<i>T2</i>	<i>T3</i>	<i>T4</i>	<i>T5</i>	<i>T6</i>	<i>T7</i>	<i>T8</i>		
Q1	1.1✓	0.7✓	2.2✓	1.8✓	1.9✓	1.1✓	3.2✓	1.4✓	1.1✓	0.6✓	1.1✓	0.6✓	1.5✓	1.9✓	0.6✓	2.6✓	5.7✓	1.1✓	1.7	
Q2	2.9✓	0.6✓	1.0✓	2.1✓	1.2✓	1.7✓	2.1✓	0.7✓	1.3✓	0.8✓	5.1✓	0.7✓	1.6✓	2.5✓	0.5✓	2.5✓	3.9✓	4.9✓	2.0	
Q3	2.0✓	1.5✓	2.2✓	2.1✓	1.3✓	5.9✓	1.0✓	3.0✓	2.3✓	1.5✓	0.9✓	1.7✓	4.0✓	1.9✓	1.0✓	1.5✓	6.6✓	5.4✓	2.5	
Q4	1.1✓	0.8✓	0.6✓	0.4✓	0.8✓	0.7✓	0.8✓	1.0✓	1.3✓	1.0✓	0.8✓	0.8✓	1.7✓	2.1✓	1.0✓	1.7✓	1.3✓	0.4✓	1.0	
Q5	1.5✓	5.6✓	3.4✓	4.2*	6.1✓	3.3✓	5.8✓	1.8✓	5.0✓	1.7✓	7.2✓	6.8*	7.2✓	5.5✓	2.8✓	7.0✓	6.5*	1.4✓	4.4	
Q6	1.0✓	0.8✓	0.9✓	0.7✓	0.7✓	1.8✓	3.0✓	0.6✓	5.0✓	1.2✓	1.7✓	0.6✓	2.9✓	1.6✓	0.8✓	1.5✓	1.6✓	3.5✓	1.7	
Q7	2.7✓	2.6✓	3.2✓	2.6✓	4.9✓	3.4✓	1.3✓	1.5✓	2.2✓	2.0✓	2.3✓	3.5✓	2.9✓	2.5✓	1.8✓	5.2✓	1.9✓	1.6✓	2.7	
Q8	1.7✓	2.1✓	4.3✓	3.1✓	—■	5.3✓	8.1*	1.2✓	3.3✓	2.0✓	6.5*	—■	5.4✓	5.3✓	2.6✓	7.8*	—■	2.7✓	3.3	
Mean	1.8	1.8	2.2	1.8	2.4	2.9	2.5	1.4	2.7	1.3	2.7	1.3	3.4	2.9	1.4	3.1	3.5	2.6	2.3	

data in the table shows that in 135 of the 144 cases (94%), participants answered the questions successfully without any hint and with a mean time of 2.3 minutes per question; in computing the mean, we did not include cases (shown in italics in Table 6.5) we considered unsuccessful in which a hint was given or the ten minutes time limit was exceeded. We consider that this data shows strong support for the usefulness of our model.

In only 23 cases (16%), participants required more than five minutes to answer the question and a hint (see Section 6.3.2) was given in nine cases (6%). In one case (S4), a hint was given before the five minutes as the subject was under time pressure to complete the study and return to a high priority work task. Two of the three subjects (T6-T8) that did not have any prior knowledge of the source control mechanism in use had the highest mean time per answered question. Even with this lack of knowledge, these participants still successfully completed most questions with a mean time of less than five minutes per question. In only three cases (2%) a participant did not reach the answer of a question within 10 minutes at which point we considered the case as a failure and asked the participant to move on to another question. The low number of hints given and unsuccessful answers support the statement that developers can use the information fragment model to answer questions they face.

To evaluate our second research question “Can developers use the model effectively without requiring a detailed understanding of how the model works?”, we consider how much information we needed to provide a participant while working on a question and how a participant used the model to answer questions. In terms of information provided to participants, we gave participants only ten to fifteen minutes of training on the model; in comparison, in a study of SQL, Chan and colleagues trained participants for an hour and a half [15]. Despite this low amount of training, for 135 cases (94%) this training was sufficient for the participants to use the model to answer questions correctly. In only 9 cases (6%), an additional hint was given by the experimenter. This low rate of hints and the low amount of time required on average, 2.3 min, to successfully answer a question, provides support for the statement that our model can be used effectively without understanding the details of the model.

We also considered how a participant used the model to answer questions. By

analyzing the screen capture videos of the participants working, we found that the participants reordered the information shown in the composition view more often (277 times) than the participants restarted the composition (213 times). Participants stated in follow-up interviews that the ability to change the projection of the composition through reordering was “definitely helpful” as “you can really do it in two steps”(S10), that “I throw everything in it [the view], reorder it [...] and see if it seems reasonable”(S2), and even that it was “great” (T8). The higher number of reorderings compared to restarts and comments of the participants provide further evidence that the participants understood enough features of the model to use it effectively without extensive training.

6.3.5 How Do Developers Use the Model?

Each of the eight questions can be answered in multiple ways with our model. To determine which aspects of the model developers use, we analyzed the combination of fragments, composition and presentation developers used to provide each answer and the process of getting there.

Variations in Answers

Participants answered the eight questions in a variety of ways. Figure 6.3 presents the number of variations for each question and how many of the eighteen participants used each variation to answer the question. The figure also shows the most commonly used answer. The less commonly used answers are listed in Table 6.6. The base fragments in the figure and the table refer to the ones presented in Table 6.4. If the base fragment has no index, any of the predefined base fragments of a domain of information leads to the correct answer and it does not matter which one is used.

For each question, there were at least two, and up to four, different variations that developers used to answer the question. For example for question Q4, twelve developers answered the question by combining the team Fragment T_1 with the work item fragment WI_2 and ordering it so that the team fragment was first ($\phi_{(T_1, WI_2)}(T_1 \otimes_{id} WI_2)$). Five developers used an inverse projection order, putting the work item fragment first ($\phi_{(WI_2, T_1)}(T_1 \otimes_{id} WI_2)$), and one developer composed

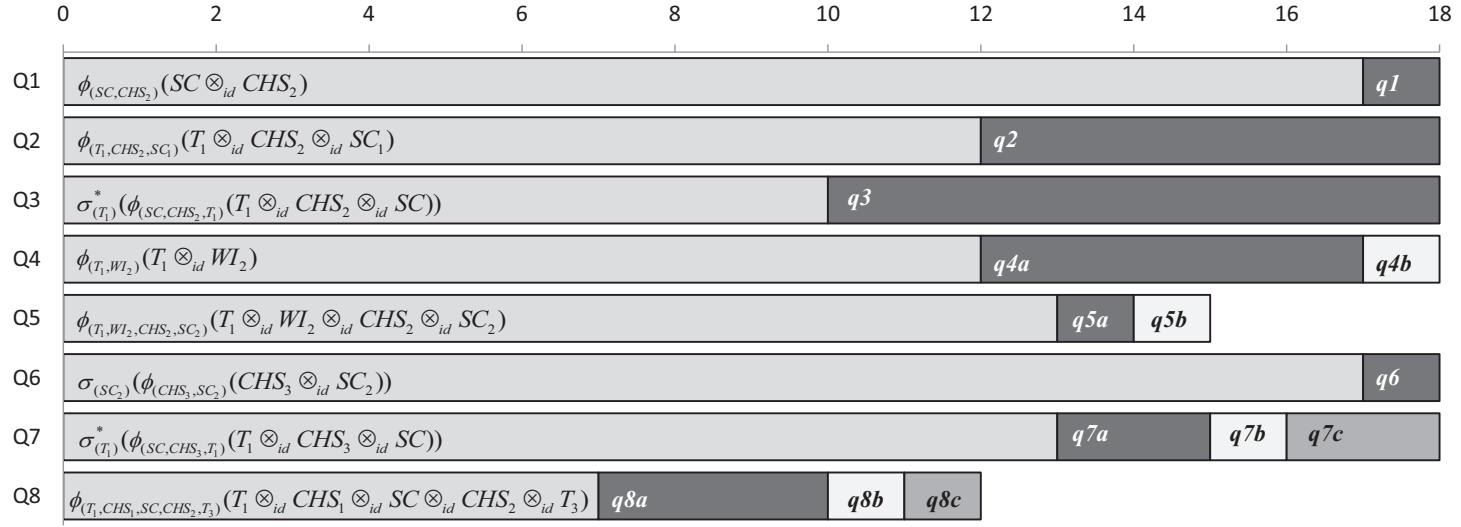


Figure 6.3: Variations in Answer over 18 Participants (bars represent number of participants that used one particular solution to a question; σ^* : count functionality that allows you to count children of an element in the tree)

Table 6.6: Answer Variations (*hidden* refers to the case in which a developer used the hide action to elide an information fragment from the presentation)

Question	Answer	Variation
Q1	q1	$\phi_{(CHS_2, SC)}(SC \otimes_{id} CHS_2)$ (CHS_2 hidden)
Q2	q2	$\sigma_{(T_1)}(\phi_{(SC_1, CHS_2, T_1)}(T_1 \otimes_{id} CHS_2 \otimes_{id} SC_1))$
Q3	q3	$\phi_{(SC, CHS_2, T_1)}(T_1 \otimes_{id} CHS_2 \otimes_{id} SC)$ (expanded/hidden parts)
Q4	q4a	$\phi_{(WI_2, T_1)}(T_1 \otimes_{id} WI_2)$
Q4	q4b	$\phi_{(T_1, CHS_2, WI_2)}(T_1 \otimes_{id} CHS_2 \otimes_{id} WI_2)$ (CHS_2 hidden)
Q5	q5a	$\phi_{(T_1, CHS_2, WI_2, CHS_2, SC_2)}(SC_2 \otimes_{id} CHS_2 \otimes_{id} WI_2 \otimes_{id} CHS_2 \otimes_{id} T_1)$
Q5	q5b	$\phi_{(SC_2, CHS_2, WI_2, T_1)}(SC_2 \otimes_{id} CHS_2 \otimes_{id} WI_2 \otimes_{id} T_2)$
Q6	q6	$\sigma(SC_2)(\phi_{(T_1, CHS_3, SC_2)}(T_1 \otimes_{id} CHS_3 \otimes_{id} SC_2))$
Q7	q7a	$\phi_{(SC, CHS_3)}(SC \otimes_{id} CHS_3)$ (expanded; change set labels contain author names in this case)
Q7	q7b	$\sigma_{(CHS_3)}(\phi_{(SC, CHS_3, T_1)}(SC \otimes_{id} CHS_3 \otimes_{id} T_1))$
Q7	q7c	$\phi_{(SC, CHS_3, T_1)}(SC \otimes_{id} CHS_3 \otimes_{id} T_1)$ (expanded/hidden parts)
Q8	q8a	$\phi_{(T_1, CHS_1, SC, CHS_2)}(T_1 \otimes_{id} CHS_1 \otimes_{id} SC \otimes_{id} CHS_2)$
Q8	q8b	$\phi_{(CHS_1, SC, CHS_2)}(CHS_1 \otimes_{id} SC \otimes_{id} CHS_2)$
Q8	q8b	$\phi_{(CHS_1, SC, CHS_2, T_3)}(CHS_1 \otimes_{id} SC \otimes_{id} CHS_2 \otimes_{id} T_3)$

an additional change set fragment CH_2 for his answer. All three variations allowed developers to answer question Q4 correctly. In general, the most common variations in answers to the eight questions were: an inverse presentation order in combination with an additional counting function (see for example answer variations for Q2); manually expanding and counting the presented information versus using the provided counting functionality (see variations for Q3); and instead of adding a team fragment to determine the author of a change, some developers just

used the information provided from the change set fragment, since the change set label contains the author name in most cases (see variations for Q8).

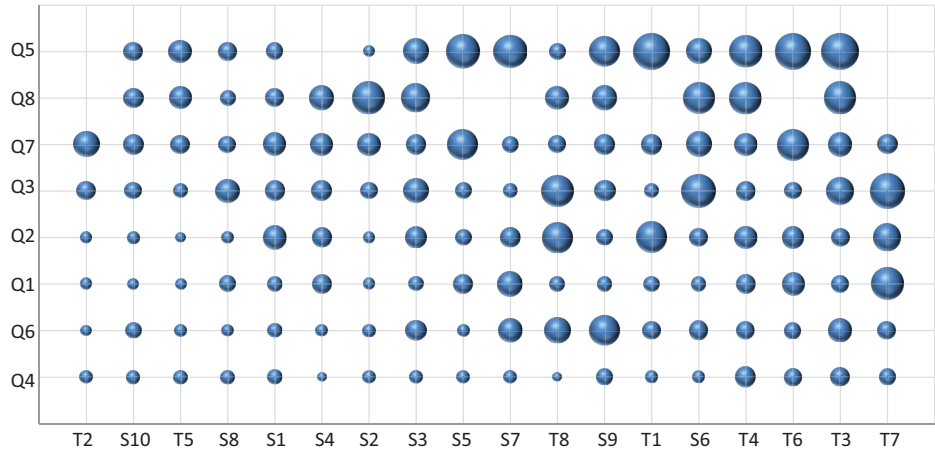
This variety in combining fragments, composition and presentation to answer the eight questions provides further evidence that the developers understood enough features of the model to use it effectively without extensive training and shows that the model supports individual preferences.

Variations in Answering Process

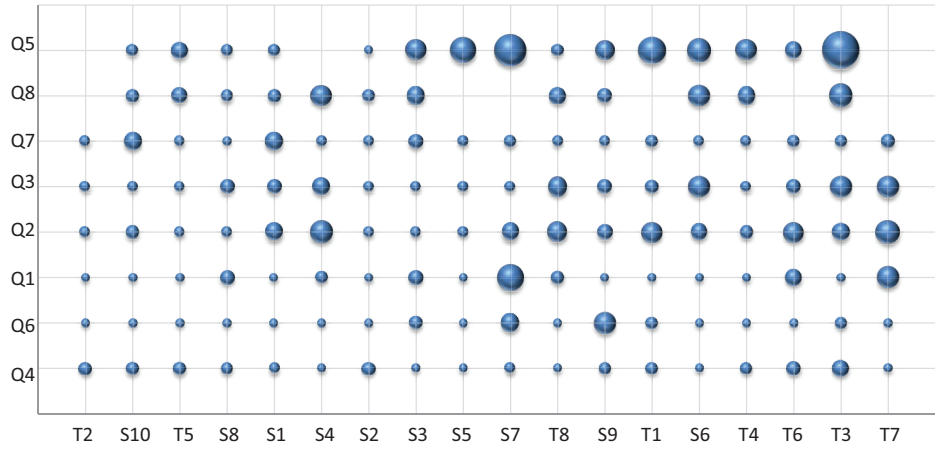
We looked at the screen capture videos of developers to analyze how they used the model in the process of answering the eight questions. In particular we were interested in how developers composed base fragments, how many times they restarted a question, how often they reordered the presentation of information and how their different usage of the tool correlated to time used to answer a question.

To identify trends, we plotted the data for each question for each subject relative to the factor of interest. For instance, Figure 6.4b presents the total number of base fragments each developer used in the process of answering a question. The bigger the bubble, the more fragments used. Developer T3, for example, used 46 base fragments in the process of answering question Q5 including restarts and only two to answer question Q1. Figure 6.4c presents the number of restarts people used in the process of answering a question and Figure 6.4d displays the number of times a developer reordered the information in the composition view to answer a question. For all four figures, developers are ordered by their mean time for answering all eight questions (faster mean time being further to the left) and questions are ordered by the mean time spent to answer it (faster answer time being closer to the bottom). If a developer did not succeed in answering a question, no bubble is displayed for that question.

To investigate the data further, Table 6.7 lists the Pearson's product-moment coefficients between the time and the number of fragments, the number of restarts and the number of reorderings used in the process of answering the eight questions. Since certain questions took considerably more time to be answered and certain developers took considerably more time to answer the questions, the table also lists the coefficients between the time and any of the other three variables, averaged over each question and over each developer.

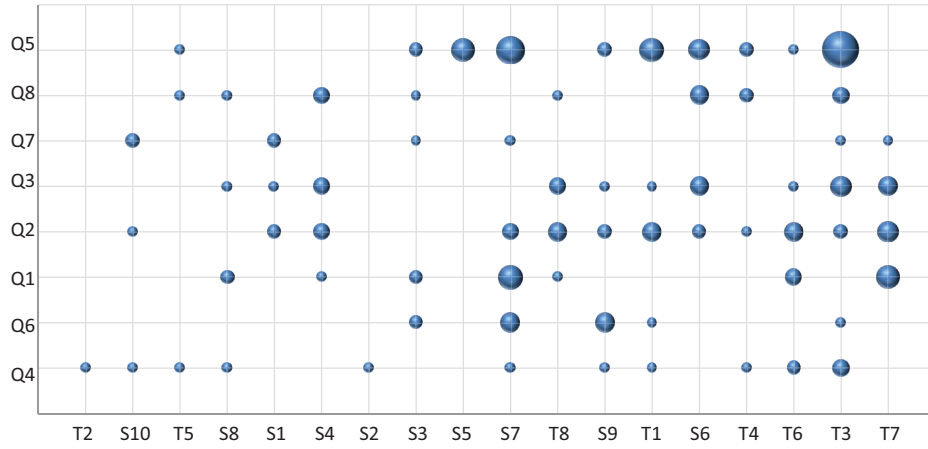


(a) Time used by Developer in Process of Answering Question (Bubble Area Reflects the Time to Complete)

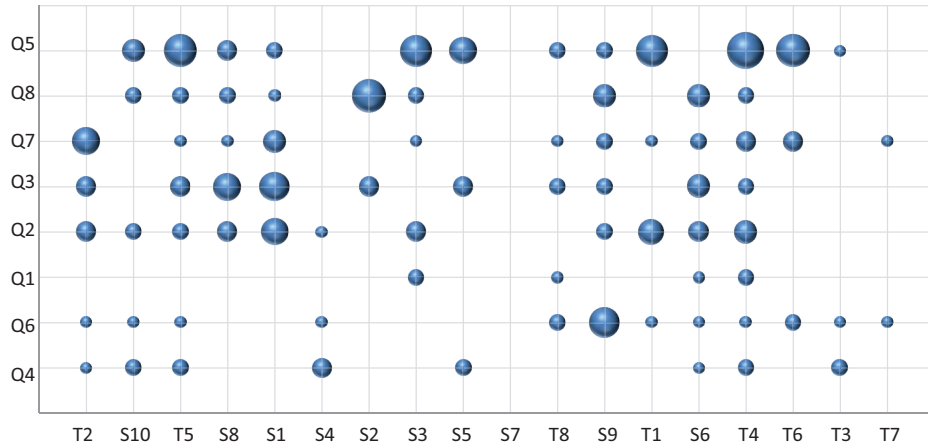


(b) Fragments used by Developer in Process of Answering Question (Bubble Area Reflects the Number of Fragments Used)

Figure 6.4: Aspects in the Process of Answering Questions



(c) Restarts used by Developer in Process of Answering Question (Bubble Area Reflects the Number of Restarts)



(d) Reorderings used by Developer in Process of Answering Question (Bubble Area Reflects the Number of Reorderings)

Figure 6.4: Aspects in the Process of Answering Questions (continued)

Table 6.7: Correlations Between Time (T) and Fragments Used (FU), Reorderings (RO) and Restarts (RS)

	mean over all	mean per question	mean per developer
T - FU	0.71	0.85	0.70
T - RS	0.64	0.71	0.70
T - RO	0.43	0.85	-0.25

Time and Number of Fragments. Both, Figure 6.4a and 6.4b show the same trend, with bubbles being bigger towards the upper right corner. A Pearson's product-moment coefficient of 0.71 supports this trend, confirming that there is a strong positive correlation between the time and the number of fragments used to getting to the answer of a question. Looking into the correlation averaged over each question or each developer still provides us the same picture. This states that the longer it took to answer a question the more fragments were used in the process of answering the question, and that the longer a developer took to answer questions, the more fragments he used in the process of answering them.

Time and Restarts. A similar trend is visible for the restarts shown in Figure 6.4c and a Pearson's coefficient of 0.64 shows that there is a strong positive correlation between time and restarts, stating that developers that took more time to answer a question also restarted more often. As for the number of fragments, the correlation over each question and over each developer provides the same picture.

Time and Reordering. No real trend is visible in Figure 6.4d and a Pearson's coefficient of 0.43 between time and reordering shows that even though there is a positive correlation, it is not strong. Averaging over each developer or each question shows a different picture. The correlation between the mean time per question and the mean number of reorderings per question is strong with a coefficient of 0.85, stating that the longer it took all developers to answer a question, the more reorderings were used by all developers in the process. The weak negative correlation between time and reorderings (-0.25) suggests that using more or less reorderings in the answering process has no impact on the time it takes a developer

to answer all eight questions. This is not surprising, as developers comments suggest that they used the reordering with different intensity. For example, developer S7 did not use reordering at all as he was “not comfortable with it” and he stated that “until I get comfortable [...], I use the basics” which for him did not include reordering. Others stated that they forgot about the reordering in the beginning but then used it more later (T1, T3), or that they needed to get used to it, but that at the end of the study it was already easier (S2, S3, S9, T1, T2, T3, T6, T7) and that the reordering “was great” (T8).

Reordering vs Restarting. Since developers commented on when they used restarting or reordering, we also looked into the correlation between these two aspects. While there is no real correlation (coefficient of 0.05) between these two aspects over all questions and subjects, the strong negative correlation (coefficient of -0.68) per developer shows that the more reordering a developer used the less restarts he used and that the more restarts he used the less reorderings he used. This reflects the varying preference amongst developers for using either of these aspects. For instance, T6 stated explicitly that “at first it felt easier to start from scratch [but] now that I understand [reordering], moving around is easier”. Developer S1 stated that sometimes a clear start is easier than reordering, and T3 said that “if there are two elements [fragments] I prefer to reorder but if there are like five elements I can’t remember how I did that so I prefer to restart”.

The overall usage of the composition of base fragments, the restarting and the reordering shows that developers were able to use the different aspects of the model to answer the questions successfully without having to understand every detail. In particular, the difference in the usage of the reordering shows that developers were able to use the model in their preferred way to answer the question.

6.3.6 What Do Developers Think About the Approach?

We analyzed transcripts of the interviews for comments related to the developer’s experience of using the approach.

Answers Multi-Domain Questions. Developers commented that the approach helps answering questions that are unsupported at the moment—“it [the tool] is answering questions that I don’t think we can answer right now” (T1). Developer S7 talked about it in more detail: “there are two links we are missing [...] and it’s nice to have something that ties them together like this”, “for this type of thing I never really thought of trying to solve these problems [...], like if I was asking what classes of this have changed in the past week you know I try and find some obscure way of comparing baselines or whatever [...] and then I’d see if there were changes to those files [...] and then I look at those change sets and browse the work items”. This comment supports our statement that the approach is supporting developers in answering questions that are difficult to answer with existent approaches.

Understanding Links Takes Time. One aspect of the user interface developers commented on was the empty view displayed when two fragments that had no edges or links between them were ordered next to each other. For example, S1, S6, S7 and T4 stated that not showing anything because there are no links between neighboring fragments is unintuitive. Others stated that it took a “lot of trial and error” (S2) to find out “which one [fragment] links” (S5), but that it “made sense in the end” (T4) and that it just takes some time to get used to it (S1,S2,S3,S9,T1,T2,T3,T4,T5,T6,T7). Developers suggested that the “linkage should be more intuitive”(T4) and that the UI could have been more explicit and show which fragments have links and which ones do not (S9,S10,T6,T7,T8).

Suggestions for the User Interface. Developers pointed out that the user interface could have been more explicit about the contents being displayed. In particular, developers commented that the icons representing the ordering of the projection were too small and subtle (S1,S6,T1,T2,T4). Furthermore, developers stated that at least one path of the tree should be fully expanded after composing a fragment, or the tool should have a visual indicator to show that the tree has more levels underneath, as it was not obvious what was in the view (T1,T2,T5). Developers S6 and T4 said that they would have liked to have a phrase summarizing the contents of the view. Another feature often commented upon was the lack in the tool’s user

interface of a way to remove a base fragment once added to the composed view. Developers said that they had to restart more often due to the lack of this feature, especially in cases where they just added a fragment by accident.

Overall Positive. When asked explicitly, all developers stated that, if available, they can imagine and would use the prototype. Some developers even stated without us asking that it was “pretty nifty”(S4), “pretty cool”(S3), “cool”(S7,S6), a “neat tool”(T2), “really really cool”(T8) and that “it is answering questions that I don’t think we can answer right now”(T1).

6.4 Threats

A threat to the validity of the study is that the set of questions might not be representative of the questions developers ask for two reasons. First, the case study only included eight of the 78 questions. We tried to minimize this threat by choosing the eight questions to cover the most common domains as can be seen from the corresponding questions shown in Table 5.1. Second, the 78 questions were identified in interviews by using a prototype of the tool as prompt which might have biased the questions found. As other research has found similar questions (e.g., [55–57]), we believe that the effect of the prompt was limited.

Another threat to the study is that the group of developers might not be representative of the population of developers. We mitigated this risk by having developers from different teams as well as different locations participating in our study.

Several threats might have influenced the time it took developers to answer questions. One threat is that some developers participating in the case study also participated in the interview sessions for finding developer’s questions. These developers saw the prototype before the case study and might have had an advantage. However, as there were six months in between the two studies and the prototype was only presented for a very short amount of time without being used by the developers, we believe that it does not have an impact on the outcome of the study.

Developers also had to identify themselves when they were done with answering a question. This assumes that the developers understand the questions correctly. As some questions were still slightly ambiguous and developers used different ap-

proaches to answering each question, this might have influenced the results.

Another threat to the findings of our study is the use of data with which the developers were unfamiliar. If the data had been of the developers own team and work, it would have been easier for the developer's to interpret the results of compositions, possibly easing the task of answering the questions.

Furthermore, three developers were unfamiliar with the source control mechanism used in our study. This might negatively impact the results for these developers. The results actually show that two of these three developers had a worse average mean time, but still completed most of the questions successfully.

Finally, we predefined the information fragments available for use in the study. As these information fragments were tailored towards the questions asked, developers may have had to spend less time than otherwise to answer a question of interest. On the other hand, these predefined fragments also made it more difficult for a developer to restrict the information shown to what he desired seeing.

Chapter 7

Discussion and Future Work

In investigating each of the two developer-centric models, we had to make a number of choices about the design of each model and how we evaluated each model. We discuss these issues for each model in turn (Section 7.1 and Section 7.2). We then discuss how these models might be combined to support developers in staying aware of relevant web feeds (Section 7.3). As part of this discussion, we also consider ways in which the work described in this thesis can be extended in the future.

7.1 Degree-of-Knowledge Model

We first review the choices we made in the two studies *Study_{EXP_DOK}* and *Study_{DATA_DOK}* to determine a developer’s knowledge of a code element (Section 7.1.1). Second, we discuss the choice we made for combining interaction and authorship in the DOK model (Section 7.1.2). In the onboarding scenario, one of the three scenarios we covered in our exploratory case studies, the DOK model did not provide any benefit. We describe possible shortcomings of the DOK model for this scenario and how we can improve the model for this scenario (Section 7.1.3). Finally, we propose directions for future research for the DOK model (Section 7.1.4 through Section 7.1.6).

7.1.1 Structural Knowledge vs. User Rating

In our exploratory study for determining factors to capture a developer’s knowledge (Study_{EXP_DOK}), we focused our investigation on knowledge of the structure of the source code and whether or not a developer can answer correctly a question about the structure of the code. We believe that structural knowledge represents an important part of a developer’s knowledge. In the later experiment we conducted to determine the relative effects of authorship and interaction (Experiment_{DOK}), we were concerned with finding a qualitative ranking of how much a developer knows about an element. Therefore, we instead asked developers in Experiment_{DOK} to rate their knowledge of each code element on a five point Likert-scale. Optimally, we would have used the same approach in each of the studies. We chose to use the qualitative approach for the second study because of limitations on developer time. Asking the developers’ to answer detailed structural questions about a large number of code elements would have exceeded the time developers had to participate. In addition, ranking the level of a developer’s knowledge based on structural questions would be difficult to do objectively.

7.1.2 Linear Regression

Our definition of a DOK model uses a linear combination of the degree-of-interest and the degree-of-authorship. The case studies we conducted provide evidence that a simple linear combination can help developers. Other combinations of the factors might provide a better fit for modeling a developer’s knowledge. Given the initial evidence from the scenarios, a more sophisticated and sensitive combinations of factors may now be warranted.

7.1.3 API Elements, Structural Information and Usage Expertise

In the onboarding case study (Section 4.1.2), using the DOK model to determine code elements a project newcomer should focus on when trying to learn about parts of the code did not provide any benefit. In this case study, developers suggested API elements as important that DOK values did not capture. The root of the problem is that API elements, by necessity, do not change often. In the three month period we considered for the authorship component of DOK when conducting this

case study, there was not a sufficient number of events on the API elements for their DOK to rise based on authorship, because the API elements were first authored and had not changed substantially in the last three months. Furthermore, as API elements often become basic knowledge, experienced developers do not need to interact with them frequently so there will be little to no interaction data to cause the DOK values to rise. The developers who participated in the case study stated that the elements for which DOK values were high are often one or two layers below the API elements. A possible improvement to the DOK model could be to propagate DOK values along structural relationships, such as subclass or call relationships, as it is likely that a developer writing code that uses or extends API elements also has some knowledge about the used or extended API element. In general, integrating structural information, such as type hierarchy and call hierarchy, into the model could help infer a developer's knowledge for closely related code elements and improve the model's performance on scenarios such as onboarding.

7.1.4 Using DOK to Prevent Bugs

Making changes to code elements a developer has little knowledge about might result in more bugs in the code changes. A DOK model could help to examine whether certain changes to the code might be more error prone than others due to the developer's lack of knowledge on the code. We are interested in conducting a retrospective study in which we examine whether change sets delivered by developers with a high degree-of-knowledge in the area of change result in less bug reports being created on the changed code and whether we might be able to use the DOK model to predict error proneness of code.

7.1.5 Finding Relevant Bugs

With the flood of information a developer faces each day through incoming emails, news feeds and other sources, mechanisms to rank and filter the incoming information can provide value to help a developer stay aware of relevant information. In one of the three exploratory case studies we conducted to determine whether the DOK model can provide value to software developers, we applied our DOK model to identify changes of interest and their corresponding bug reports for a group of

three developers (Section 4.1.3). In future work, we intend to extend our study onto more teams and extend the model to incorporate information elements other than just source code, such as bug reports. Similar to code, bug reports have a first author, have changes made by many developers as comments are added, and have interactions performed on them by developers who read and consult the reports. A DOK model might be helpful in prioritizing which bug reports a developer should consider first.

7.1.6 Longitudinal Study

Since there are many individual and project specific factors that might influence indicators for a developer’s knowledge, a longitudinal study that captures different phases of a project could provide additional insight of how a DOK model might be improved. Such a study could also provide more evidence on whether or not a general DOK model is robust enough to handle most situations or whether different project situations require different models.

7.2 Information Fragments Model

The three major components of the information fragments model are the information fragments, the composition and the presentation. We first describe how the composition in our approach is automated (Section 7.2.1), before discussing the text matching operator (Section 7.2.2). Second, we compare the composition and presentation to a query language (Section 7.2.3). We then talk about the selection of information fragments (Section 7.2.4) and how we extended our prototype by another domain of information (Section 7.2.5). Finally, we review possible presentations (Section 7.2.6).

7.2.1 Automatic Composition

Using the information fragments model, a developer can focus on indicating the portions of information from the project in which he is interested. Once the information of interest is determined, the composition of the different portions of information is done automatically.

For 71 of the 78 questions in study *Study_{EXP-FR}*, only the identifier matching

operator is required to answer the question. For all of these 71 cases, the operator can be chosen and will find the necessary identifier matches automatically based on the properties of the nodes. The remaining seven questions are 68 to 72, 54 and 77 in Table 5.1. Five of these remaining questions (68 to 72) require information from the web site domain to be composed with other information using the text matching operator. Since these are the only five questions of the 78 that require web site information, the composition operator can be automatically chosen by always just applying text matching for this type of fragment. To answer the question “What are the emails related to line items and defects that are features?” (54), the text matching operator is needed to match emails to line items using the subject line in the email. For this question, the text matching operator can also be chosen automatically since the identifier matching operator does not apply for these two kinds of information, email and line item. Finally, for the question “Which conversations in work items have I been mentioned [in]?” (77), the text matching operator is needed to match team member names in comments. Applying the identifier matching operator to compose the comment and team information fragment will result in comments being matched with their authors. Applying both composition operators automatically for this question, as well as for the questions that also require fragments of comments and of teams in the same order (75 and 76) answers the question properly. Only some additional information that the user might not want to see is displayed based on the identifier matching operator. For instance in case of the question 77, the developer would not only see which conversations he is mentioned in but also which conversations he authored. However, since the presentation of edges in the composed viewer includes a rationale for the edge between information nodes, such as a team member is the *author* of a comment, the user can easily distinguish relevant from irrelevant information.

7.2.2 Text Matching and Other Operators

As described in Section 5.3, different to matching identifiers, text matching introduces a certain fuzziness in the result to account for variations in the textual match, such as word synonyms or spelling mistakes. Different developers might prefer text matching with different levels of confidence on the text match. To support

confidence thresholds, the model needs to be adapted to either provide different operators or to adapt the threshold of the edges presented in the integrated view. In addition to variations in the text matching, there might be other operators that developers want to answer additional questions. Whether or not future operators can also be chosen automatically is an open question. Another open question is how to handle cases in the future in which more than one composition operator applies. For now, the composition operators can be chosen automatically. However, with new composition operators, such as text matching operators with different confidence thresholds, several composition operators might apply at the same time. Allowing the developer to choose the composition operators puts more burden on the developer, while limiting the choice decreases the flexibility and the number of possible questions that can be answered.

7.2.3 Model vs SQL

Our information fragments model is less expressive than general query languages such as SQL [14, 19]. We can map the composition operators in our model to inner joins in SQL and the presentation operators to the order or aggregate functions in SQL. With the information fragments model, we intentionally chose not to provide the full power of a query language. We made this choice to allow us to automate the composition and therefore reduce the complexity, while still providing the flexibility to answer all 78 questions according to the information needs of the developers.

7.2.4 Information Fragment Selection

In our study of the prototype tool we built to support the information fragments model, we relied on predefined fragments and focused our attention on the composition and presentation of information fragments. Although we have enhanced existing views in RTC to provide simple support for user-selected fragments, an open research question is how to best support a developer in selecting fragments of interest. It may be possible to predefine fragments for such a large number of questions to make sophisticated selection mechanisms unnecessary. Most information fragments required to answer the 78 questions (Section 6.1) varied only

with respect to time, for instance WI_1 through WI_3 , or CO_1 and CO_2 in Table 6.1. Providing a set of predefined fragments together with an option to choose the time frame of the fragment, such as a time slider, e.g., for choosing the last day, the last week or the last month, may be sufficient. We leave this investigation to future work.

7.2.5 Extending the Tool

Several developers stated that they would like the information fragments model to be extended to other types of information. For instance, developers wanted it to provide support for answering questions on test cases, such as “Which changes caused the tests to fail and thus the build to break?” (63 in Table 5.1). To evaluate the generality of our approach, we extended our tool with test case information. The major component of the work was in defining the properties for test cases and determining how to retrieve this information within the development environment. No changes were required in the composition operators or the overall model. Only locally constrained and straightforward changes were required, suggesting that our approach is easy to extend for other kinds of information.

7.2.6 Presenting the Information

An early prototype tool we developed to support the model provided a more graph-like presentation of the composed information. However, through conversations with developers, we found that developers often prefer tree-based views; in part because they want to avoid using up extra screen real estate typically required of graph-like presentations. Since the focus of our study was to evaluate whether or not developers can apply the model to answer the questions they face, we restricted our prototype to a tree view. In the interviews after our case study, several developers suggested more sophisticated presentations based on pivot tables or graph-like presentations due to their higher expressiveness. Due to the separation of the presentation from the other aspects of the model, extending our approach with different presentations is straightforward. We leave the investigation of the possible benefit from different presentations for future work.

7.3 Using Knowledge and Context for Awareness

The degree-of-knowledge and the information fragments model can be synergistically used together, for instance, for project awareness. To stay aware of relevant information in a project, developers often subscribe to information streams, such as RSS feeds. However, not all of the information in a stream is relevant. Finding the relevant information within the vast amount of information a developer faces each day is difficult, in particular, since the information is presented without context and the developer has to manually relate it to his work.

The case study on identifying interesting change sets presented earlier (Section 4.1.3) provides evidence that the DOK model can be used to identify relevant information. In addition, by supporting the composition of fragments of various kinds of information, the information fragments model provides context that can ease the task of staying aware. For instance, using the developer's workspace as a context to rank and sort the information in a stream, might allow the developer to quickly determine which items in the information stream affect or reference the code in the workspace that are of relevance to him [28]. By combining the DOK model with the information fragment model we might be able to provide relevant information in context to support a developer in staying aware of feeds.

Exploratory Study. To investigate the value of using the two models together, we conducted an exploratory study, which we refer to as *Study_{FEEDS}*, on determining relevant news items in feeds (see [30] for more details on this study and its results). Our study involved five professional developers that used IBM's Rational Team Concert (RTC) in their daily work. For this study, we interviewed each developer two times. In each interview, we presented the participant with a list of 30 random news items from default feeds in RTC that captured changes to the code and to work items (see Figure 7.1a). We asked each participant to review the items and tell us whether or not the item was relevant. In the second round of interviews we used the information fragments model to present the news items in the context of the source code on which the developer's team worked (see Figure 7.1b) and also in the context of the team. In the case of source code context, if a news item could be related to the source code, the news item was shown hierarchically beneath the related source code (respective, in the case of team context, the related

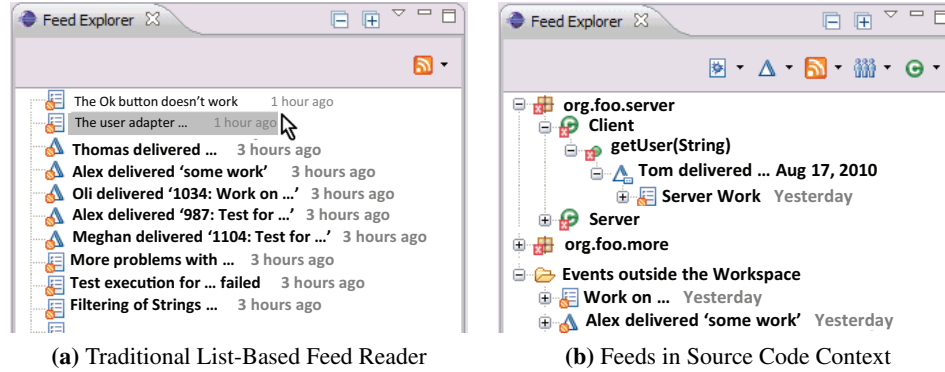


Figure 7.1: Presentation of Feeds.

team member). If one of the news items did not relate to the source code (respective, the team), it was shown in the by default flat list. We asked the participant whether presenting the news in terms of context changed their determination of the relevance of an item and whether or not the context was helpful. We chose to investigate source code and team context as developers described them as important based on our interview-based study about which questions developers ask (Section 5.1). For each developer, we also collected the interaction history over the last seven workdays and the authorship information over the past three months needed to calculate the DOK value of the element.

Each interview session took between 20 and 60 minutes. All interviews were recorded and the interviewer (author of this thesis) took handwritten notes. Due to the exploratory nature of our study, we parsed our data using an open coding technique to develop and identify categories of data [79].

Providing Context With the Information Fragments Model. From the 150 news items that we presented to the five developers in the second session, only 37 (25%) were easily determined to relate to the code on which the participants worked and presented in the context of the source code. In 20 of the 37 (54%) cases, participants considered the source code context to be helpful. In 9 of 37 (24%) cases the participants referred to the context as being somewhat helpful and in 8 (22%) cases as being not helpful. In particular, when the short form of the summary used to identify the news item was too generic, such as “delivered 2 change sets”, the

context provided a significant benefit to the participant in explaining the item.

When asked about source code context in general, three out of the five participants considered the placement of news items in context helpful. One of the three stated that he “had problems with the [news items] out of context, but [for] the ones within, [he could determine relevancy] pretty quick”. Another participant preferred the team context, stating that the source code context provided too much information but that the team context allows skimming over people quickly. Three more participants conceived the team context as helpful, in particular in situations in which you do not have to work with a lot of source code, such as quality control.

Using the DOK Model to Identify Relevant News Items. We think that combining the DOK model with the information fragments model can provide valuable support to help a developer identify relevant items of information. In our case study on identifying interesting change sets presented earlier (Section 4.1.3) we showed how the DOK model can support the developer. To stay aware of relevant web feeds, we think that we can apply a similar approach to identify relevant feed items. Instead of just presenting feed items in context using the information fragments model, we could use the information on which feed items are relevant to rank and maybe filter the presented information.

To investigate whether DOK values provide value in the exploratory study (Study_{FEEDS}) and can indicate whether or not a news item is relevant, we looked at source code elements related to the news item and calculated the DOK values for each of these source code element with respect to the interviewed developer. It turns out that for only one of the five developers was a single news item related to source code elements with a positive DOK value. This news item was considered as relevant by the developer. However, a single data point does not provide any evidence. All other 29 items in that interview as well as the items used in all other interviews with the five developers did not relate to a source code element with a positive DOK value. We believe that the lack of related source code elements with positive DOK might be due to the fact that we conducted the study at the end of an iteration cycle with less coding activity as well as the completely random selection of the news items. Further study is needed to support our hypothesis on the combination of the two models to support a developer’s awareness.

Chapter 8

Conclusion

During a workday, a software developer must continuously search for the small portions of information pertinent to his work within the flood of project information. Today's artifact-centered development environments make finding the needed information tedious or infeasible.

In this thesis, we have introduced the idea of a developer-centric model that helps support the developer in accessing the needed information. Two developer-centric models were introduced and investigated: the degree-of-knowledge (DOK) model and the information fragments model.

The degree-of-knowledge (DOK) model provides a means of describing which part of a code base each developer knows. This model takes an individual perspective on a developer's knowledge of code. A developer's degree-of-knowledge is based on two components: the developer's authorship of the code and the developer's interaction of the code. Two empirical studies we conducted, Study_{EXP_DOK} and Study_{DATA_DOK}, showed that both components, interaction and authorship, are important to capture a developer's knowledge of code. Study_{EXP_DOK} showed that, with statistical significance, a developer's interaction with code can be used as an indicator for the developer's knowledge of the code. Study_{DATA_DOK} provided evidence that authorship plays an important role in a developers knowledge of the code and that authorship and interaction capture two different aspects of a developer's knowledge.

Using the DOK model, questions such as who knows which code and who

should know about a change can be supported. To evaluate the benefits and limitations of our approach, we conducted case studies with professional developers. For answering who knows which code, we conducted case studies on expert finding at two different sites and provide evidence that the DOK model outperforms traditional approaches. For the question who should know about a change, we have shown in a different case study, how our DOK model might be used to pick out changes of interest in the environment to the developer. Despite substantial variations in the authorship and interaction behavior between different development sites, the DOK model is generic enough to perform well at the different sites and thus robust enough to be applied across different sites.

The combination of authorship and interaction in our model allows us to better capture the ebb and flow in a developer's knowledge than existing approaches solely based on authorship and to take into account one shorter- and one longer-term aspect of a developer's knowledge. Despite the simplicity of the linear regression and the model itself, the degree-of-knowledge model provides value in different scenarios that a developer faces. Extending the model with more structural information as well as increasing the granularity of the approach, such as taking into account the number of lines being changed rather than just looking at the element level, could help to improve the accuracy of the model and better capture API elements.

The information fragment model supports the composition of information from multiple sources and the presentation of the composed information in flexible ways. In an interview-based study with eleven professional developers, we identified 78 questions that developers ask and that span across different domains of information together with the answers the developers desired for the questions. We showed that the information fragments model can support all of these 78 questions with respect to the developers' desired answers. We also showed that a prototype tool can choose the composition operators to combine this information automatically.

Just because a model is expressive does not mean it is usable. To show that developers can use the model easily and effectively, we conducted a study with 18 professional developers. These developers were able to successfully answer a high percentage (94%) of questions posed in minimal time with little training. The information fragment model is an approach to balance expressivity with simplicity.

While it is expressive enough to support the 78 questions we identified in the interviews with the developers, it supports the questions by automatically composing the information fragments easing the composition of the information done by the developer. The automatic composition allows a developer to avoid specifying how different information fragments have to be linked. The separation of composition from presentation in the information fragments model allowed the developers to tailor the composed information to their personal needs.

These two models show that it is possible to add developer-centric models to a development environment and ease a developer's access to the information relevant to work-at-hand addressing the developer's individual information needs. We have discussed how these two models might even be synergistically combined to support a developer in staying aware of relevant information in a project, helping him to rank and filter the new information in his working context. Directions for future work lie in applying both models to other scenarios as well as other kinds of software artifacts.

Bibliography

- [1] http://www.eclipse.org/eclipse/development/eclipse_3_0_stats.html. → pages 1
- [2] <http://www.w3.org/standards/semanticweb/>. → pages 20
- [3] E. M. Altmann. Near-term memory in programming: a simulation-based analysis. *International Journal of Human Computer Studies*, 54(2):189–210, 2001. → pages 15
- [4] G. Antoniol, M. D. Penta, H. C. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, 127(3):87–99, April 2005. → pages 20
- [5] L. M. Berlin. Beyond program understanding: A look at programming expertise in industry. In C. R. C. Jean C. Scholtz and J. C. Spohrer, editors, *Proc. of the Fifth Workshop on Empirical Studies of Programmers*, pages 6–25. Ablex Publishing Corporation, 1993. → pages 15
- [6] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001. → pages 20
- [7] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM. doi:<http://doi.acm.org/10.1145/1240624.1240823>. → pages 4, 18
- [8] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM. doi:<http://doi.acm.org/10.1145/1718918.1718973>. → pages 16

- [9] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press. → pages 4, 14
- [10] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983. doi:D01:10.1016/S0020-7373(83)80031-5. → pages 14
- [11] R. R. Burton, L. M. Masinter, D. G. Bobrow, W. S. Haugeland, R. M. Kaplan, and B. A. Sheil. Overview and status of doradolisip. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP'80, pages 243–247, New York, NY, USA, 1980. ACM. → pages 11
- [12] R. H. Campbell and P. A. Kirsliis. The saga project: A system for software development. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 73–80, New York, NY, USA, 1984. ACM. → pages 11
- [13] N. R. Carlson, W. Buskist, M. E. Enzle, and C. D. Heth. *Psychology: the Science of Behaviour*. Prentice Hall Canada, 2005. → pages 14
- [14] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. → pages 126
- [15] H. Chan, K. Siau, and K.-K. Wei. The effect of data model, system and task characteristics on user query performance: an empirical study. *SIGMIS Database*, 29(1):31–49, 1997. doi:http://doi.acm.org/10.1145/506812.506820. → pages 81, 107, 109
- [16] D. H. Chau, B. Myers, and A. Faulring. What to do when search fails: finding information by association. In *Proc. of CHI '08*, pages 999–1008, New York, USA, 2008. ACM. doi:http://doi.acm.org/10.1145/1357054.1357208. → pages 17
- [17] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *Eclipse'03: Proc. of the 2003 OOPSLA eTX Workshop*, pages 45–49, New York, NY, USA, 2003. ACM. doi:http://doi.acm.org/10.1145/965660.965670. → pages 19

- [18] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proc. of CHI'07*, pages 557–566. ACM, 2007.
doi:<http://doi.acm.org/10.1145/1240624.1240714>. → pages 3, 6, 53, 54, 59
- [19] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970. → pages 126
- [20] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proc. of ICSE'08*, pages 21–30, New York, USA, 2008. ACM.
doi:<http://doi.acm.org/10.1145/1368088.1368092>. → pages 16, 19
- [21] R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 207–210, New York, NY, USA, 2010. ACM.
doi:<http://doi.acm.org/10.1145/1810295.1810331>. → pages 7, 19
- [22] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proc. of SoftVis'05*, pages 183–192. ACM, 2005.
doi:<http://doi.acm.org/10.1145/1056018.1056044>. → pages 14
- [23] F. Détienné. *Software design—cognitive aspects*. Springer-Verlag New York, Inc., New York, NY, USA, 2002. → pages 14, 23
- [24] P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie: a knowledge-based software information system. In *Proc. of ICSE'90*, pages 249–261, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. → pages 18
- [25] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, New York, NY, USA, 1992. ACM. doi:<http://doi.acm.org/10.1145/143457.143468>. → pages 6, 53
- [26] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.
doi:<http://dx.doi.org/10.1109/32.177365>. → pages 7, 19

- [27] K. Erdos and H. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 98 –105, 1998. doi:10.1109/WPC.1998.693322. → pages 16
- [28] T. Fritz. Staying aware of relevant feeds in context. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 523–524, New York, NY, USA, 2010. ACM. doi:http://doi.acm.org/10.1145/1810295.1810462. → pages 128
- [29] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184, New York, NY, USA, 2010. ACM. doi:http://doi.acm.org/10.1145/1806799.1806828. → pages 17
- [30] T. Fritz and G. C. Murphy. Determining relevancy: How software developers determine relevant information in feeds. Technical Report TR-2010-12, University of British Columbia, December 2010. → pages 128
- [31] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer’s activity indicate knowledge of code? In *ESEC-FSE'07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM. doi:http://doi.acm.org/10.1145/1287624.1287673. → pages 23
- [32] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 385–394, New York, NY, USA, 2010. ACM. doi:http://doi.acm.org/10.1145/1806799.1806856. → pages 43, 54
- [33] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE'04: Proc. of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society. → pages 19
- [34] D. Garlan. Views for tools in integrated environments. In R. Conradi, T. Didriksen, and D. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 314–343. Springer Berlin / Heidelberg, 1986. doi:http://dx.doi.org/10.1007/3-540-17189-4_105. → pages 18

- [35] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. of IWPSE'05*, pages 113–122. IEEE Computer Society, 2005. doi:<http://dx.doi.org/10.1109/IWPSE.2005.21>. → pages 12
- [36] D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, 2007. → pages 86
- [37] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, New York, NY, USA, 2004. ACM. doi:<http://doi.acm.org/10.1145/1031607.1031621>. → pages 4
- [38] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Trans. Softw. Eng.*, 12:1117–1127, December 1986. → pages 2, 11
- [39] E. Hajiyeve, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In D. Thomas, editor, *Proc. of ECOOP'06*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer. → pages 18
- [40] R. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160, 1950. → pages 86
- [41] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. *Mining Software Repositories, International Workshop on*, 0:141–150, 2009. doi:<http://doi.ieeecomputersociety.org/10.1109/MSR.2009.5069492>. → pages 12
- [42] J. D. Herbsleb and E. Kuwana. Preserving knowledge in design projects: what designers need to know. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems, CHI '93*, pages 7–14, New York, NY, USA, 1993. ACM. doi:<http://doi.acm.org/10.1145/169059.169061>. → pages 16
- [43] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proc. of MSR'08*, pages 23–26, New York, USA, 2008. ACM. doi:<http://doi.acm.org/10.1145/1370750.1370755>. → pages 18
- [44] R. Holmes and R. J. Walker. Customized awareness: recommending relevant external change events. In *Proceedings of the 32nd ACM/IEEE International*

Conference on Software Engineering - Volume 1, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM. → pages 6, 53

- [45] M. Jakobsen, R. Fernandez, M. Czerwinski, K. Inkpen, O. Kulyk, and G. Robertson. Wipdash: Work item and people dashboard for software development teams. In *Human-Computer Interaction INTERACT '09*, volume 5727 of *Lecture Notes in Computer Science*, pages 791–804. Springer, 2009. → pages 18
- [46] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of AOSD'03*, pages 178–187, New York, NY, USA, 2003. ACM. doi:<http://doi.acm.org/10.1145/643603.643622>. → pages 17
- [47] D. Jin and J. R. Cordy. Ontology-based software analysis and reengineering tool integration: The oasis service-sharing methodology. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 613–616, Washington, DC, USA, 2005. IEEE Computer Society. doi:<http://dx.doi.org/10.1109/ICSM.2005.68>. → pages 7, 20
- [48] D. Jin and J. R. Cordy. Integrating reverse engineering tools using a service-sharing methodology. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 94–99, Washington, DC, USA, 2006. IEEE Computer Society. doi:<http://dx.doi.org/10.1109/ICPC.2006.30>. → pages 20
- [49] W. L. Johnson and A. Erdem. Interactive explanation of software systems. *Automated Software Engineering*, 4:53–75, 1997. doi:<http://dx.doi.org/10.1023/A:1008655629091>. → pages 16
- [50] M. Kersten. *Focusing knowledge work with task context*. PhD thesis, University of British Columbia, 2007. → pages 26, 44
- [51] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD'05: Proc. of the 4th International Conference on Aspect-Oriented Software Development*, pages 159–168, New York, NY, USA, 2005. ACM. doi:<http://doi.acm.org/10.1145/1052898.1052912>. → pages 13
- [52] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT'06/FSE-14: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, New York, NY, USA, 2006. ACM. doi:<http://doi.acm.org/10.1145/1181775.1181777>. → pages x, 4, 13, 25, 26, 27, 44, 48, 61

- [53] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparql and a software evolution ontology. In *ICSEW '07*, Washington, DC, USA, 2007. IEEE Computer Society.
doi:<http://dx.doi.org/10.1109/ICSEW.2007.139>. → pages 7, 20
- [54] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32:971–987, 2006.
doi:<http://doi.ieeecomputersociety.org/10.1109/TSE.2006.116>. → pages 4, 15
- [55] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of ICSE'07*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
doi:<http://dx.doi.org/10.1109/ICSE.2007.45>. → pages 1, 6, 17, 79, 119
- [56] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Second Workshop on the Evaluation and Usability of Programming Languages and Tools at SPLASH '10*, 2010. → pages 3, 6, 17, 53
- [57] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of ICSE'06*, pages 492–501, New York, NY, USA, 2006. ACM.
doi:<http://doi.acm.org/10.1145/1134285.1134355>. → pages 3, 6, 17, 53, 79, 119
- [58] S. Letovsky. Cognitive processes in program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp. → pages 16
- [59] M. A. Linton. Implementing relational views of programs. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, New York, NY, USA, 1984. ACM. doi:<http://doi.acm.org/10.1145/800020.808258>. → pages 18
- [60] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240, New York, NY, USA, 2000. ACM Press.
doi:<http://doi.acm.org/10.1145/358916.358994>. → pages 3, 6, 12, 53

- [61] P. Mi and W. Scacchi. A meta-model for formulating knowledge-based models of software development. *Decision Support Systems*, 17(4):313 – 330, 1996. doi:DOI:10.1016/0167-9236(96)00007-3. → pages 20
- [62] S. Minto and G. C. Murphy. Recommending emergent teams. In *Proc. of MSR'07*. IEEE Computer Society, 2007. → pages 12
- [63] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM. doi:http://doi.acm.org/10.1145/581339.581401. → pages 3, 12, 38, 58
- [64] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006. doi:http://dx.doi.org/10.1109/MS.2006.105. → pages 27
- [65] E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM. doi:http://doi.acm.org/10.1145/1879211.1879216. → pages 14
- [66] C. Parnin, C. Görg, and S. Rugaber. Enriching revision history with interactions. In *Proc. of MSR'06*, pages 155–158. ACM, 2006. doi:http://doi.acm.org/10.1145/1137983.1138019. → pages 14
- [67] S. Paul and A. Prakash. A query algebra for program databases. *IEEE Trans. Softw. Eng.*, 22(3):202–217, 1996. doi:http://dx.doi.org/10.1109/32.489080. → pages 7, 18
- [68] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295 – 341, 1987. doi:DOI:10.1016/0010-0285(87)90007-7. → pages 14
- [69] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *Proc. of ISWC'03*, volume 2870/2003, pages 738–753. Springer, 2003. → pages 21
- [70] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988. doi:http://dx.doi.org/10.1109/2.86782. → pages 37

- [71] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society. → pages 4, 18
- [72] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 94–103, New York, NY, USA, 2007. ACM. doi:<http://doi.acm.org/10.1145/1321631.1321647>. → pages 4
- [73] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proc. of MSR'08*, pages 121–124. ACM, 2008. doi:<http://doi.acm.org/10.1145/1370750.1370779>. → pages 13
- [74] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21:96–101, May 2006. doi:<http://dx.doi.org/10.1109/MIS.2006.62>. → pages 20
- [75] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proc. of FSE'06*, pages 23–34, New York, NY, USA, 2006. ACM. doi:<http://doi.acm.org/10.1145/1181775.1181779>. → pages 16
- [76] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97*, 1997. → pages 1
- [77] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984. → pages 15
- [78] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988. doi:<http://doi.acm.org/10.1145/50087.50088>. → pages 1
- [79] A. C. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Inc, 2nd edition, 1990. → pages 129
- [80] W. Teitelman and L. Masinter. The interlisp programming environment. *IEEE Computer*, 14(4):25–33, 1981. doi:<http://dx.doi.org/10.1109/C-M.1981.220410>. → pages 11

- [81] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005. doi:<http://doi.ieeecomputersociety.org/10.1109/TSE.2005.71>. → pages 7, 18
- [82] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of 16th International Conference on Software Engineering*, pages 39–48, 1994. → pages 4, 14
- [83] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 165–174, New York, NY, USA, 2010. ACM. doi:<http://doi.acm.org/10.1145/1806799.1806827>. → pages 18
- [84] Y. Ye, Y. Yamamoto, and K. Nakakoji. A socio-technical framework for supporting programmers. In *Proc. of ESEC-FSE'07*, pages 351–360, New York, NY, USA, 2007. ACM. doi:<http://doi.acm.org/10.1145/1287624.1287674>. → pages 18
- [85] L. Zou and M. W. Godfrey. Understanding interaction differences between newcomer and expert programmers. In *Proc. of RSSE'08*, pages 26–29. ACM, 2008. → pages 14

Appendix A

Studies: Supporting Materials

This chapter provides a table (Table A.1) that lists all studies conducted for this thesis and the supporting materials for the studies.

A.1 Sample Questions for Study_{EXP_DOK}

1. How much and what kind of experience do you have with this code base?
2. How long did you work on this part (or a particular code element) before the monitoring period?
3. Are you the author of this part of the code?
4. Why do you think you know this part of the program well?
5. Is there a particular reason why you answered this question (question from the questionnaire) the way you did?
6. Did your responsibilities for parts of the code base change during the study and how?
7. Do you think the questions in the questionnaire are appropriate and adequate for measuring your knowledge and why or why not?

Table A.1: Studies Conducted for this Thesis

Study Name	Study Description	Supporting Material
Study _{EXP_DOK}	Formative study to investigate the factors in modeling knowledge	Sample questions (Section A.1)
Study _{DATA_DOK}	Data study to investigate authorship and interaction behavior of developers	<i>none</i>
Experiment _{DOK}	Experiment to determine the weighting factors for the factors in the DOK model	<i>none</i>
Study _{CS_DOK}	Case studies to evaluate the DOK model	<i>none</i>
Study _{DATA.2.DOK}	Data study to investigate the general applicability of the DOK model	<i>none</i>
Study _{EXP_FR}	Interview study to investigate the range of questions that span across multiple kinds of information	<i>none</i>
Study _{INFR}	Case study to evaluate the usefulness of the information fragments model	Tutorial (Section A.4), sample questions (Section A.2)
Study _{FEEDS}	Exploratory study on determining relevant news items in feeds	Sample questions (Section A.3)

A.2 Sample Questions for Study_{INFR}

1. How would you use our tool to answer the question "Who is working on what"?
2. In which situations and for which questions can you imagine using our tool?
3. How much and what kind of experience do you have with this code base?
4. What does the information presented in the view mean to you?
5. Is there any information missing in the view that you want to see to answer the question at hand?
6. Are there any features missing that you would like the tool to have?

A.3 Sample Questions for Study_{FEEDS}

1. What is the reason for an item of information to be relevant?
2. Are there any particular rules you are using to determine whether an information item is relevant or not?
3. When do you consider an item of information as irrelevant?
4. Does the context provided in the Awareness View help you to interpret the information or rather make it more difficult?
5. Is there any information missing in the view that you want to see to stay aware of relevant information?
6. Is the presentation in the Awareness View well-suited for staying aware of relevant information?
7. Are there any features missing in the Awareness View that you would like it to have?
8. Can you imagine using the Awareness View and in which situations?
9. With respect to the two views: which one was easier to use for you and why?

A.4 Tutorial for the Evaluation of the Information Fragments Model

The *Fragment Explorer* is a view inside of Jazz that allows you to compose different kinds of information, such as Java source code, change sets, work items and teams. We use the term *fragment* to refer to a subset of a kind of information, such as all change sets created yesterday or all members of your team.


The Fragment Explorer has two parts. The lower part (marked A in the attached screenshot) lists the fragments you can select to compose. The upper part of the view (marked B in the attached screenshot) shows the result of composed fragments. We refer to the upper part as the *composed viewer*.

For the remainder of this session, please assume that you are Alex, one of 9 developers on the Source Control Mechanism team for Jazz (March 18th, 2008). The code for your team and any dependent code is loaded into your workspace. Imagine that you have just arrived at the office in the morning and you have a couple of questions about the development. Please follow the steps as described below!


Question #1: “Which changes have been delivered in the last week by the Jazz team and by whom have they been delivered?”


To answer this question,

- Select the fragment “Change sets delivered in the last week” and press the “+” Button (marked C in the attached screenshot) to add the fragment to the composed viewer. You can also right click on the fragment and press “Compose Selection” or double click on the fragment. (Please do so now.)

Adding the fragment will display the elements of the fragment as a tree. For now, the tree has one level and thus appears as a list. Also, a button with an icon  representing the added change set fragment will be added on the top right of the composed viewer that lets you perform additional actions on the representation of the added fragment (marked D in the attached screenshot).

- Next add the fragment “Jazz team” to the composed viewer by selecting it and pressing the “+” Button (marked C in the attached screenshot). The newly added fragment is composed with the fragment that was already in the composed viewer. As change sets have information about the team member that authored the change, the composition creates links between change sets and team members. The viewer now displays the intersection. So if you, for example, expand the first change set node, you can see that Todd from the Work Item and the Agile Planning team delivered changes on March 12th. (Note: change sets that have been delivered by developers that are not on the Jazz team will not be displayed!)

Adding the fragment will again result in an icon/button being added in the top right corner of the view (marked D) that represents the team fragment . The order of the icons represents the order of the fragments in the common

viewer. Right now the order is   meaning that change sets come first and team elements second in the composed viewer.

These two steps answer the first question.



Question #2: “Which member of the Jazz team created most changes in the last week?”

To answer this question,


- Use the “Count... > Contributors” (or the “Count Team Elements”) action of the drop down menu of the button representing the team fragment in the top right corner (marked D in the screenshot). You will see a table view appear as a separate window that summarizes the occurrences of each of the Jazz team members (and the teams). In this case Matt made most changes (77). Note that the count view will summarize the occurrences of each element on that level in the composed viewer. Changing the order of fragments in the composed viewer can change the occurrences of an element in the viewer and thus the occurrence count in the count view.

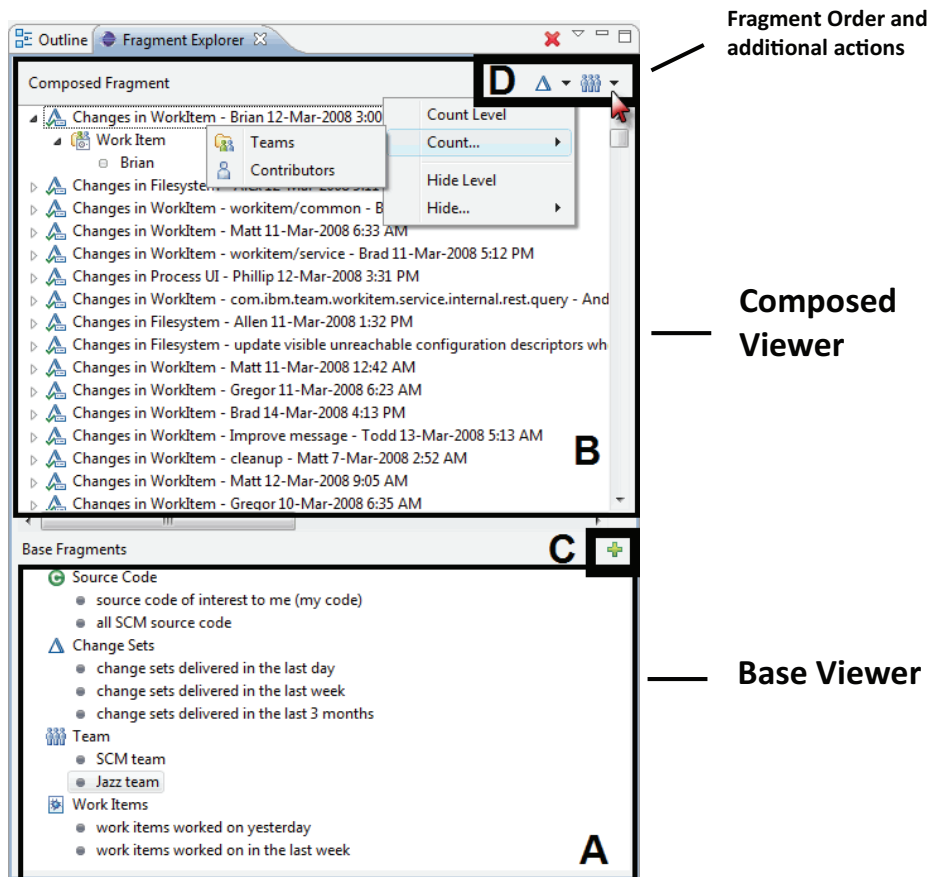
Question #3: “Which team created which change sets in the last week?”

To answer this question,



- Change the order of the presentation in the composed viewer by changing the order of the fragment icons using drag and drop on the top right of the view (marked D). The order should look like   afterwards so that team elements come before change sets.

NOTE: As the composition shown in the composed viewer is commutative, the order in which you add the fragments does not have any effect on the underlying composition. Changing the fragment order adapts the presentation but does not change the composed information.

- As you are only interested in the teams and not the members, you can use the “Hide... > Contributors” action in the drop down menu of the Team fragment button . This action shows only the teams and the change sets they



created in the last week underneath.

The drop down menus  and  provide actions on the elements in the view of the previously added fragments (team and change set respectively).

Now please clear the composed view by pressing the red X button in the top right corner of the view (just above D).