

Master Thesis

September 14, 2013

Identifying a Starting Context of Code Elements for a Change Task

Katja Kevic

of Nesslau-Krummenau, Nesslau SG, Switzerland (07-709-850)

supervised by

Prof. Dr. Thomas Fritz



University of
Zurich^{UZH}



Master Thesis

Identifying a Starting Context of Code Elements for a Change Task

Katja Kevic



**University of
Zurich** UZH



Master Thesis

Author: Katja Kevic, Katja.Kevic@uzh.ch

Project period: 15.03.2013 - 15.09.2013

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank all the people who were involved in this thesis. First of all, I thank professor Thomas Fritz for giving me the opportunity to write this thesis, for the great ideas and for the great support. Many thanks also to David Sheperd who supported me in the execution of the exploratory study and many thanks to all people which participated in the study.

Abstract

Developers spend a substantial amount of their time searching, navigating and reading source code while performing a change task. Several research approaches have been suggested to recommend points in the source code, which are relevant for the change task at hand. However, none of these approaches leverages information given through the Mylyn task context.

We conducted an exploratory study to investigate this initial search, reading and navigation phase before the source code is changed. We found that presenting a context along with search results, supports a developer in selecting relevant search results. Furthermore, we identified that a major challenge during the initial search phase, is the finding of good search terms given the change task description.

To overcome the challenge of finding good search terms for a change task at hand, we suggest two approaches which leverage information given through the Mylyn task contexts. The design implications on the presentation of the search result context, are implemented in a prototype application which automatically suggests starting points for a change task at hand by using user interaction histories.

Zusammenfassung

Software Entwickler verbringen während der Bearbeitung einer Source Code Änderungsaufgabe erheblich viel Zeit damit, den Source Code zu durchsuchen, zu navigieren und zu lesen. Verschiedene Forschungen haben sich damit auseinandergesetzt, relevante Stellen im Source Code für die Änderungsaufgabe dem Software Entwickler vorzuschlagen. Keine der Forschungen hat sich jedoch mit dem Einsatz eines Mylyn task contexts beschäftigt.

Wir führten eine explorative Studie durch, die sich mit der anfänglichen Suche und Navigation im Source Code auseinandersetzte, bevor der Software Entwickler die Änderung im Source Code vornimmt. Wir fanden heraus, dass ein Kontext der Suchresultate die Software Entwickler unterstützt um relevanten Suchresultate näher zu betrachten. Ausserdem haben wir das Finden eines guten Suchbegriffs als eine der grössten Schwierigkeiten identifiziert.

Um das Finden eines guten Suchbegriffes zu erleichtern, schlagen wir zwei Vorgehen vor. Beide dieser Vorgehen nützen den Mylyn task context aus. Des Weiteren haben wir Implikationen der Präsentation von einem Kontext der Suchresultate in einem Prototypen umgesetzt. Der Prototyp schlägt basierend auf Benutzerinteraktionen Stellen im Source Code vor, die die Bearbeitung einer Änderungsaufgabe erleichtert.

Contents

1	Introduction	1
1.1	Structure	2
2	Related Work	3
2.1	Source Code Comprehension for Performing a Change Task	3
2.1.1	Using Dynamic Information to Understand Source Code	4
2.1.2	Using Static Information to Understand Source Code	4
2.1.3	Using Hybrid Information to Understand Source Code	5
2.1.4	User Development Interaction	5
2.1.5	Conclusion on Related Work on Source Code Comprehension	6
2.2	Exploratory Studies on how a Developer Performs a Change Task	6
2.3	Search Term Recommendation Systems	7
3	Exploratory Study with Developers	9
3.1	Method	9
3.1.1	Participants	9
3.1.2	Sando	10
3.1.3	Comogen	10
3.1.4	Change Tasks	10
3.1.5	Tools and Instrumentation	13
3.1.6	Procedure	14
3.2	Study Results	14
3.2.1	Browsing Code Structures to Find Task Relevant Points	15
3.2.2	Browsing for Task Related Recommendation Results	17
3.2.3	Finding Good Search Terms	18
3.2.4	Qualitative Analysis of the Semi-Structured Interview	19
3.3	Threats to Validity	20
3.3.1	External Validity	20
3.3.2	Internal Validity	20
3.3.3	Construct Validity	21
3.3.4	Statistical Validity	21
3.4	Conclusion on the Exploratory Study	22
3.5	Implications from the Explorative Study	22
3.5.1	Browsing Code Structures to Find Relevant Points	22
3.5.2	Browsing for Task Related Recommendation Results	23
3.5.3	Finding Good Search Terms	23

4	Using Interaction Histories to Recommend a Starting Context of Code Elements for a Change Task	25
4.1	Finding Similar Change Tasks	26
4.1.1	Finding Similar Change Tasks to a New Change Task	26
4.1.2	Finding Similar Change Tasks for a Re-opened Change Task	30
4.2	Recommending Potentially Relevant Starting Points	30
4.2.1	Generation of Starting Points	31
4.2.2	Generation of a Context	33
4.3	Conclusion	33
5	Two prototypes for Automatically Recommend Search Terms for a Change Task	35
5.1	Approach 1 for Recommending Search Terms	35
5.2	Approach 2 for Recommending Search Terms	37
5.3	Prototypical Implementation	38
5.4	Case Study	39
5.5	Discussion	40
6	Approach: Automatic Identification of a Starting Context	43
6.1	Usage Scenario	43
6.2	Identification of a Starting Context	44
7	Implementation Details	47
7.1	Data Model	47
7.2	Starting Context Recommender	47
7.3	View	49
7.4	Connector	49
7.5	Source Code Parser	49
7.6	Similarity Engine	49
7.7	Search Term Recommender	50
7.8	Visual Studio Monitor	50
8	Future Work	53
8.1	Presenting the Rationale of a Recommendation	53
8.2	Remembering Relevant Search Results	53
8.3	Including Selections of Prior Searches in the Query	53
8.4	Examining Task Contexts	54
9	Conclusion	55
A	Participant Instructions	57
B	Questionnaire	59
C	Interview quintessences	61
D	Text Preprocessing	63
E	Contents of the CD-ROM	67
F	Used Libraries, Tools and Plug-ins	69

List of Figures

3.1	A screen shot from the tool Sando. The search term <i>split</i> is used to query the code base. For the search term <i>split</i> 40 recommendations of points in the source code are suggested. The first result is expanded, such that the relevant code snippet is visible. With a double-click on any result one can jump into the source code.	11
3.2	A screen shot from the tool Comogen. The search term <i>log</i> is used to query the code base. For the search term <i>log</i> 6 recommendations of points in the source are suggested. With a double-click on any method call one can jump into the source code.	12
3.3	Sketch of a possible representation of the recommendation results along with a context.	24
4.1	We identify the features which match in change task with a similar task context. Then we apply this pattern to a set of change tasks to determine change tasks which cover a similar concept.	27
4.2	To investigate if user interaction patterns are also reflected in the task contexts, we analyzed the interactions immediately before a particular interaction and the interactions immediately after.	33
4.3	Our approach generates recommendation for starting point based on task context elements of a similar change task.	34
5.1	Our first approach to automatically recommend search terms includes the identification of a similar change task. Within the term corpus on source code level of the concept, we identify terms which describe the new change task best.	36
5.2	The concept of our second approach. We assume that terms of the natural language term corpus match one or more terms of the source code term corpus. A set of natural language terms defines a concept of a change task, which can be mapped to a set of terms in the source code term corpus. As example, <i>nl-t1</i> , <i>nl-t11</i> and <i>nl-t12</i> define the concept <i>c2</i> , which is expressed as <i>src-t13</i> , <i>src-t5</i> and <i>src-t22</i> in the source code.	38
5.3	The search term recommender views. 1.) is a screenshot of the user entries and 2.) is a screenshot of the result view which includes the suggested search terms.	39
6.1	The workflow of our approach to automatically recommend a starting context for a change task at hand. First, similar change task are determined through a feature pattern. Then, using the degree-of-interest model, relevant elements of a task context are pitched on. These elements are the recommendations for potentially relevant starting points for the change task at hand. Finally, we complement the recommendations with structural information of the source code.	44
6.2	To use our tool, the user has to provide the id of the change task to investigate, the url to the change task repository and the path to the local git repository. If all information is provided, the user can click the button <i>Get Recommendations</i> to generate suggestions for an initial starting context. In this screenshot one queries an initial starting context for change task 340622.	45
6.3	The recommended starting points for change task 340622. One can jump into the recommendations by double-clicking the result.	46
7.1	Overview of the five main components included in the architecture of our prototypical implementation.	48
7.2	The tools included in our Eclipse plugin.	51

D.1	The sequence in which the text preprocessing steps are accomplished.	64
-----	--	----

List of Tables

3.1	The six change tasks of the Sando project investigated by the ten participants. . . .	13
3.2	The results of the Mann-Whitney test on all the dependent variables. Comogen and Sando differ significantly in terms of navigation steps accomplished, number of queries accomplished and the number of jumped in results.	16
3.3	The numbers of relevant recommendations generated by both tools of the first search terms, the numbers of all relevant recommendations generated through all search terms after the first input and before the last input and the numbers of relevant recommendations generated with the last search terms.	19
4.1	The combination of features for both initial sets together with the number of matched change tasks (true positives and false positives regarding the concepts covered by the pairs) out of 36 change task from the project org.eclipse.mylyn.context.	28
4.2	The LOF caption	29
4.3	We identified six different index term selection strategies which are used to calculate similarity measures between text features of change tasks.	29
5.1	The four change tasks of the project org.eclipse.mylyn.context for which we automatically generate search terms.	41
5.2	The number of results generated by the Eclipse File Search tool using the best combination of the suggested search terms and the number of results which point to methods which were changed within a change set which is linked to the change task.	42
C.1	The quintessences of the semi-structured interview (translated from German to English) we conducted with each participant after he finished working on all the task.	62

Introduction

Performing change tasks¹ is a frequent activity in the evolution of software [VvMS99]. Evidence shows that while working on change tasks, developers spend a lot of the time on comprehending the program through reading and navigating source code [KAM05], [MJS⁺00]. Ko et al. [KMCA06] reported in an explorative study, conducted with ten participants, that developers spend about 35% of their time reading code, performing searches and navigating code. Further, Paul et al. [PPBH91] specify, that software engineers spend as much time reading source code than programming code and Singer et al. [SLVA97] identified the exploration of code as an important activity in their examination of software engineering work practices. In particular, a developer is interested in locating the points in the code where the change has to be made [ASGA12]. A common process to do so is to use text search tools such as `grep`, `egrep`, `fgrep`, `ed`, `sed`, `awk` or `lex` [MRB⁺05] to locate starting points for code examination [SCH98]. This strategy may be very fast and easy in cases where the software engineer has a broad knowledge about the system, as it requires exact matching of keywords. However, identifying good search terms is not always as easy. Since, for example, the change tasks are usually filled out by the users of the software system which use a different vocabulary as the software engineers [ESW06].

Several research approaches have tried to better support developers in locating points in the source code which are related to a change task at hand (e.g. [WGGS92] and [Zha06]). They range from interactive to automated approaches mainly investigating execution traces of the program (e.g. [WGGS92] and [EDV05]) or features of the change task with the according changes in the source code (e.g. [MSRM04] and [Zha06]). However, to the best of our knowledge none of the approaches leverages information given through the Mylyn task context. In addition, even if the approaches identify code elements, that are close to the code that needs to be changed, developers still spend a lot of time navigating source code to investigate the adjacencies of the recommended elements [KDV07]. To capture this notion, Murphy et al. [MKRv05] and Ko et al. [KAM05] introduced similar concepts which consist of various elements and their relationships, which are relevant to perform a given change task—a task context.

Several studies have looked into what is interesting for a developer while comprehending source code for performing a change task. Some studies [KM05b], [VvMS99], [FKS⁺08] revealed that developers use the structure, intent and behavior of elements in the code to construct a context, and others investigated which questions arise while performing a change task [SMDV08], [SD-VFM05], [ES98]. Neither of these studies has looked into the challenges which arise when sophisticated code search tools are used to perform a change task.

In this work, we are looking at the challenges which arise in the initial search and navigation phase before an actual change is made, when a sophisticated code search tool is used. In partic-

¹The term change task is used in this work interchangeably for bugs and other modification tasks

ular, we investigate whether a context along with the recommendations, generated by a search tool, supports developers in identifying a starting point for code exploration for a given change task. Furthermore, we investigate whether we can automatically recommend starting points for a change task at hand by using developer interaction histories. What we refer to as 'starting point' is an element within the source code that is either structurally or conceptually related to the change task at hand.

This thesis contributes an exploratory study with ten developers, to investigate the initial search and navigation phase before a change is made in the source code. We found that presenting a context of the recommended starting points supports the developers in the initial search and navigation phase when performing a change task. We hypothesize that the reduced interactions with the system origin from better assessing search results when a context is given. These results are based on the observation that developers using a search tool, which presents context, navigate through significantly less source code and conduct significantly less searches. Furthermore, we observed a behavioral pattern, which occurs most frequently when developers use a search tool, which presents context to the search result. The exploratory study also showed that finding a good search term for a change task description, is a major challenge during the initial search and navigation phase. We imply possible design decisions and tool suggestions based on these results. In particular, we suggest two approaches for recommending search terms given a change task description and a representation of the search results which include a context.

In addition, this work contributes case studies on an open source project. We identified a pattern of features to find similar change tasks to a change task at hand, once by using interaction histories and once by using change set information. The found patterns perform equally in terms of preciseness. We also showed that user interaction histories, combined with the degree-of-interest model [KM05a] recommend relevant starting points for a change task at hand. We combined our findings in a prototype application which automatically identifies a starting context for a change task at hand.

1.1 Structure

This thesis is structured as follows. In section 2 we discuss research related to source code comprehension when performing a change task and review exploratory studies conducted to explore how developers perform change tasks. Section 3 summarizes the exploratory study we conducted and we discuss the results of the study and infer design and tool implications from the findings. Further, section 4 includes case studies on an open source project, to explore if user interaction histories can be used in the process of identifying a starting context for a change task at hand. We present our approach of automatically identifying starting contexts in section 6 and the implementation details of it in section 7. In section 9 we conclude our work on automatically identifying starting contexts for a change task at hand.

Related Work

This chapter discusses first two categories of research which are related to our approach of an automatic identification of a starting context and then looks into approaches which suggest search terms for a change task to query the source code.

For research related to our approach of automatically identifying a starting context we look in a first step into research which addresses the understanding of source code which needs to be changed when performing a change task (section 2.1). Specifically, we discuss research, which leverages interaction histories of a developer performing a change task, to recommend potentially relevant points in the source code (Section 2.1.4). In a second step, we look into studies which have been conducted to capture the behavior of software engineers while performing a change task (section 2.2). Section 2.3 reviews approaches which recommend search terms that can be used to query the implementation of a concept in a change task.

2.1 Source Code Comprehension for Performing a Change Task

Since many years research addresses the difficulty of understanding source code in order to perform a change task. To understand how a specific feature of the program is implemented, one can either follow a systematic strategy or follow an as-needed strategy [LPLS86]. While the systematic strategy requires the understanding of the global system behavior, the as-needed strategy focuses on local system behavior and includes solely the understanding of the components necessary to change in the source code. Since the systematic strategy is infeasible for the majority of software systems, the following focuses only on the techniques that support the as-needed strategy. Research which supports the understanding of source code in terms of the as-needed strategy can broadly be categorized into dynamic, static and hybrid approaches. The dynamic approaches (Section 2.1.1) investigate execution traces of the program to infer additional knowledge about the source code while the static approaches (section 2.1.2) analyze features of the change task and the source code to draw conclusions. Hybrid approaches (section 2.1.3) combine dynamic and static approaches. These approaches follow either an interactive or an automated strategy. The interactive approaches demands that the user interacts with the tool to locate source code which is relevant for the change task, while the automated techniques manage to locate relevant points in the source code without continuous user interactions [EDV05]. Recently, the notion of a task context— various elements and their relationships which are relevant to perform the change task—was introduced by Murphy et al. [MKRv05] and Ko et al. [KAM05]. Task Contexts support developers to better perform a change task, as it is one possible way to transfer knowledge about a feature. This section gives an overview of the main approaches, tools and techniques aimed

to locate relevant points in the source code. None of these approaches combine elements of Mylyn task contexts with conventional techniques to recommend potentially relevant points in the source code.

2.1.1 Using Dynamic Information to Understand Source Code

Dynamic bug localization techniques observe components in the source code while a feature is executed to locate its most relevant parts in the source code [ESW06]. Over 20 years ago Wilde et al. [WGS92] introduced the first dynamic bug localization approach. They found a probabilistic formulation and a deterministic formulation to map source code components of test case executions to a collection of features to locate relevant parts in the source code. There are several enhancements of this approach. Wong et al. [WHGT99] locate components at different granularity levels that are unique to features or common to a group of features using execution slices. Eisenberg and De Volder [EDV05] determine Dynamic Feature Traces. Using heuristics like multiplicity, specialization and depth enables to determine a source code component's relevance to a feature expressed as a gradual ranking. In order to provide a more comprehensive view of the implementation of a feature Wong et al. [WGH00] introduced metrics which describe the relationship between a feature and component. The metrics disparity, concentration and dedication try to quantitatively capture their closeness. These approaches imply implicitly that a test case executes in a clearly defined time interval and that the behavior of the system is deterministic. These two assumptions do not hold for distributed systems. To tackle these difficulties Edwards et al. [ESW06] come up with an approach which observes the execution of the system while noting time intervals in which components are active.

2.1.2 Using Static Information to Understand Source Code

Static concept localization techniques leverage static information from change tasks, changesets and source code. Unlike the dynamic concept localization techniques do the static concept localization techniques not necessarily require executable code. In practice, the most popular tools to locate concepts in source code are using pattern matching. The tools `grep`, `egrep`, `fgrep`, `ed`, `sed`, `awk`, and `lex` treat the source code as character stream and return lines of it which match a pattern which was specified by the user [MRB⁺05]. While these lexical tools are very versatile, easy to use, fast [MN96] and integrated with the editing environment [GYK01], they do not take into account the structure of the source code [MN96] and the performance of these tools is highly dependent on the query [MRB⁺05]. Some more elaborated tools based on pattern-matching, e.g. [MN96], [ESS92], [GYK01] counteract these flaws.

Approaches which analyze program component dependencies, e.g. [FTAM96], [MTO⁺92], [RM02], [CR01], [BMW94], are usually interactive and present a graph to the user which nodes are source code entities and which edges are the relationships between these entities. These graphs present the user structural information of the program and can be filtered, searched and clustered such that developers are supplied with possible starting points for further investigations.

Extensive research has been conducted on leveraging information retrieval techniques to better understand source code and find points in the code relevant to a feature. These approaches differ mainly in the specific techniques applied for preprocessing activities, indexing units, similarity measure definitions and the granularity of the results which are typically presented in a ranked list [MRB⁺05]. Exemplary approaches of this category are [MSRM04], [YF02], [ACCDL00], [TSL03], [MBK91] and [Zha06]. Another important approach presented by Weiser [Wei81], finds relevant components of a feature by reducing the source code to a program slice. The program slice includes all components in the source code which are possibly dependent to the feature.

These components are filtered through data and control dependencies.

2.1.3 Using Hybrid Information to Understand Source Code

Some approaches combine static and dynamic information artifacts to locate interesting points in the code. Eisenbarth et al. [EKS03] present a semi-automated approach that exploits the dynamic technique of Wilde and Scully [WS95] to obtain a feature-unit map. The results are refined afterwards manually by inspecting the dependency graph of the system. Antoniol et al. [AG05] create through static analysis of the source code a model. The relevant elements of a feature are abstracted with data gathered from dynamic analysis procedures. Rilling and Karanth [RK01] introduced a hybrid slicing algorithm, which reduces the number of program executions while maintaining the accuracy of a dynamic slicing algorithm. The hybrid approach basically prepares through a static slicing algorithm the inputs for the dynamic slicing algorithm.

2.1.4 User Development Interaction

Mylyn is an Eclipse plug-in that monitors interaction events of a developer while performing a change task. Mylyn is tightly integrated with Eclipse and enables a task-focused user interface. The task context of the change task being worked on is formed by interaction events that are classified as selections, edits, commands, preferences, predictions, propagations, manipulations and attentions. All of these interaction events are captured along with a unique identifier of the element being interacted with, timing information and an interest rate to the respective change task [Fou13]. The interest rate of the element being interacted with is based on a degree-of-interest model. This degree-of-interest model is based on specifications of interest increasing rules and interest decreasing rules while considering temporal occurrences. As example, if a developer selects a code element, the interest rate increases (+1 by default). On the other hand, the interest rate decreases, if the developer does not edit or select that code element for a specified period of time or other code elements are interacted with (-0.1 by default). Users of Mylyn can also manually add or remove elements from their current task context or delete the whole task context if needed [KM05a]. Basically, Mylyn captures a program slice from the developer's perspective. This kind of program slice supports developers when resuming a change task since the monitored task contexts can be attached to change tasks. Interesting code elements are immediately presented and serve as initial starting points for code exploration. If only little work has been performed on a change task, the prediction interaction events can be used. Mylyn task contexts are a mean of transferring and storing knowledge about interesting code elements of a change task explicitly.

Fewer research has been conducted on investigations about using task contexts from the project history to recommend potential starting points for investigations given a new change task. Although using task contexts for recommendations seems promising, as not only the final work product is considered, but also the work in between. Rastkar and Murphy [RM09] compared in a case study the change tasks pairs which have similar change sets to the change task pairs which have similar task contexts. They conclude that the change task pairs found through similar change sets are not necessarily the same change task pairs found through similar task contexts. Robillard and Manggala [RM08] present the tool ConcernDetector, which detects, depending on the current changes, related concern mappings out of a concern pool. The concern pool stores a collection of history concern mappings. Concern mappings can be created differently, as example by a technique which automatically infers concern descriptions out of program investigation activities [RM03]. However in the case study presented the tool ConcernMapper was used, which

allows the developer creating concerns manually by selecting code elements. The concern detection is based on the number of overlapping elements between the current changes and the concern mappings in the concern pool. Matched concern mappings from the concern pool are immediately recommended to the developer as a mean to decrease the effort of finding relevant code pieces while working on a change task. One approach, Team Tracks [DCR05], points developers to interesting locations in the source code by mining navigations gathered from other developers working on the project. Team Tracks offers, based on the gathered navigation data, a filtered class view and view which presents related items to a selected item. A recent approaches is implemented in the tool NavClus [LK11], which recommends clusters of code elements relevant to the changes currently made in the source code. The recommendations are based on prior interaction histories of the project. The relevancy of a code element is determined based on two principles : 1) a code element which is frequently visited during work on a change task, is likely to be highly relevant and 2) the neighboring code elements to a relevant code element are likely to be relevant as well.

2.1.5 Conclusion on Related Work on Source Code Comprehension

Even though object-oriented code ideally reflects one singular concept per class and several design patterns are introduced to support modularization, the localization of source code to change for a change task cannot be cut down to the trivial selection of the class implementing the concept [MRB⁺05], [MKRv05]. The widespread of functionality is not necessarily caused by a bad system structure. It is rather inevitable if a structure of the system cannot be integrated into another structure of the system [MKRv05]. To this end, several approaches have been proposed to support developers in better understanding source code. None of these approaches exploits information of Mylyn task contexts to automatically recommend potentially interesting locations in the source code given a new change task. Further, many of these approaches present a ranked list to the users, which includes the recommendations of locations in the source code, which are potentially relevant to a change task. Observations from our study (see section 3) claim that a developer understands the recommended results better if they are presented within a context.

2.2 Exploratory Studies on how a Developer Performs a Change Task

There have been several studies conducted to discover how developers understand source code in order to perform a change task. In the following we discuss the most influential ones and the ones which are similar to our study. One important study carried out by Ko et al. [KMCA06], [KAM05] was performed with ten participants who were told to solve five change tasks. The study aimed to better understand how a developer decides what is relevant for a given change task, what types of information is considered relevant and how developers remember locations in the code that were potentially relevant. The results inspired the authors for a new task context model which is based on searching, relating and collecting of code elements that the developer thinks are relevant. Vans et al. [VvMS99] started to explore developers' activities during change task performances very early. An experimental study with professional software engineers doing a debugging task was conducted. The focus of investigation aimed to understand better what type of actions developer perform, on which level of abstraction developers are working and how developers confronted hypothesis during debugging tasks. They showed that developers work on

all levels of abstraction while they only care what a specific piece of code does and less care why a specific piece of software is designed in a certain way.

The research most related to our research is the study conducted by Starke et al. [SLS09]. In their laboratory study, ten participants solved a change task using common Eclipse search features, such as *Open Type*, *Find in File*, *References*, *Implementor*, *Declaration*. They addressed the question of how developer decide what to look for and how developers decide which results are relevant to their query. Starke et al. observed that developers rarely look at more than one search result and start a new query instead. Our study differs insofar that we explore challenges which occur when skimming search result produced by state-of-the-art search tools, which present more sophisticated results than the Eclipse search functionalities provide.

The presented studies give precise insights on how a developer performs a change task. What we in contrary investigate are the challenges a developer faces when performing a change task using state-of-the-art search tools.

2.3 Search Term Recommendation Systems

Ko et al. [KMCA06] found in their exploratory study that most developers start working on a change task by performing firstly a textual search against the source code with a term they think describes the concept of the change task best. This section discusses research which has been conducted to support the developers in the very first step of performing a change task, namely the formulation of the query which can be entered in a code search tool. The Sando Code Search Tool ¹ employs instant feedback to the user as she begins to type a query term. These feedbacks consist of element names of the source code which match the terms being typed in and terms which are likely to be added to the provided term. Further, if the term entered does not match any element in the source code, synonyms of the term are suggested. The tool Find-Concept [SFH⁺07] supports the developer in choosing efficient query terms in an interactive process. Different terms are assigned to a Verb-Query or a Direct-Object-Query by letting the user select recommended words for adding to the Verb-Query set or to the Direct-Object-Query set. However, we are not aware of any research which leverages task contexts to automatically recommend efficient search terms for a change task at hand.

¹<http://sando.codeplex.com/>

Exploratory Study with Developers

This section describes an exploratory study conducted with ten participants working on six different change tasks. This study aims to identify challenges developers face in the initial search and navigation phase before an actual change is made [KMCA06]. Further, we want to understand whether a context along with the recommendations, generated by a search tool, support developers in identifying a starting point for a new change task. The participants investigate the change tasks either using a code search tool which presents the recommendations in form of a ranked list or a code search tool which employs a more sophisticated visualization techniques which also presents limited context information to the user. Our goal is to use the gained knowledge from our observations to elaborate design implications for a new approach of automatically recommending a starting context.

3.1 Method

We asked ten developers to investigate six change tasks of a program they were unfamiliar with. For each change task, they were asked to find three places in the source code (at least on method level) they would point a new programmer to, who will be assigned this task to work on. They were given a maximum of ten minutes to work on each task, but were allowed to finish earlier. We employed an A/B testing method with an additional counterbalance measure design to avoid bigger learning effects. Specifically, one group of the participants investigated the first three change tasks with Sando (see section 3.1.2) and the last three change tasks with Comogen (see section 3.1.3) and one group investigated the first three change tasks with Comogen and the last three change tasks with Sando.

The study was arranged in our lab such that the participants could work on the same change tasks under the same conditions. This enabled us to compare the work results better as if the participants would have investigated change tasks on a different code base as it would have been the case in a field study. It would have been impossible to predict if variations in behaviors would have been caused by different conditions such as example task difficulty or the source code quality.

3.1.1 Participants

Ten male undergraduate and graduate students from our school participated in this study. They all have a profound background in object-oriented programming (mean of 7.4 years, whereat the

shortest experience is 4 years and the longest experience is 13 years) and all of them also have professional programming experience (mean of 2.93 years, whereat the shortest experience is 0.8 years and the longest experience is 6 years). The participants were split about equally regarding their object-oriented programming experience into two groups. One group has a mean of 6.6 years of object-oriented programming experience and one group has a mean of 8.2 years of object-oriented programming experience. All of them were familiar with common navigation features, such as *Go To Definition*, *View Call Hierarchy* or *Find All References* and all of them were unfamiliar with Sando, Comogen and the source code of Sando.

3.1.2 Sando

The participants investigated change tasks of the Sando project ¹. Sando is implemented as Visual Studio Extension. It is a code search tool which leverages static information of the source code. When a user starts to type a query into Sando, it suggests various related query terms to the user which are retrieved out of the source code. After executing the query, Sando presents a ranked list of search results to the user. Each search result can be quickly explored by clicking it once and a little code snippet of relevant code to the search result is displayed. By double-clicking the search result, the user jumps directly into the source code (see figure 3.1 for a screenshot of Sando). The participants investigated six different revisions of Sando. Each of these revisions is the revision before the change task was worked on. For change task 1, the participants worked on revision 1083 which contains 5.57k lines of code. The revision 1057 for change task 2 contains 5.52k lines of code, the revision 1042 for change task 3 contains 6.71k lines of code, the revision 1902 for change task 4 contains 5.61k lines of code, the revision 779 for change task 5 contains 6.85k lines of code and the revision 1162 for change task 6 contains 5.71k lines of code ². The Code Metrics Results tool within Visual Studio Ultimate 2012 rates the Sando source code as good maintainable. The participants did not receive any introduction of the architecture of Sando.

3.1.3 Comogen

Comogen is a Visual Studio extension which uses information retrieval techniques and structural information from the code base to recommend potentially relevant points in the source code for a search term. Comogen marks each recommendation with a bar which indicates the estimated relevancy of the recommendation. Comogen includes in each recommendation result a context which displays part of the control flow which surrounds the recommendation result. The context is displayed as sequence diagram indicating relevant method calls. Several visualization adjustments can be conducted, such as zooming in the sequence diagram. To jump in directly into the source code, one has to click on a method call. See figure 3.2 for a screenshot of Comogen.

3.1.4 Change Tasks

Each participant investigated six real change tasks of the Sando project. The summary and the description of these six change tasks originated from one of the main contributors of Sando. Table 3.1 summarizes the change tasks we used in this study. These change tasks cover different parts of the functionality within Sando. For each change task, we looked up the actual changed lines of code and referred to the methods containing these changed lines of code, as the most relevant

¹<http://sando.codeplex.com/>

²The tip revision of Sando can be cloned with `https://hg.codeplex.com/sando`. With `$ hg update -r<revision>` the tip version can be updated to the specified revision.

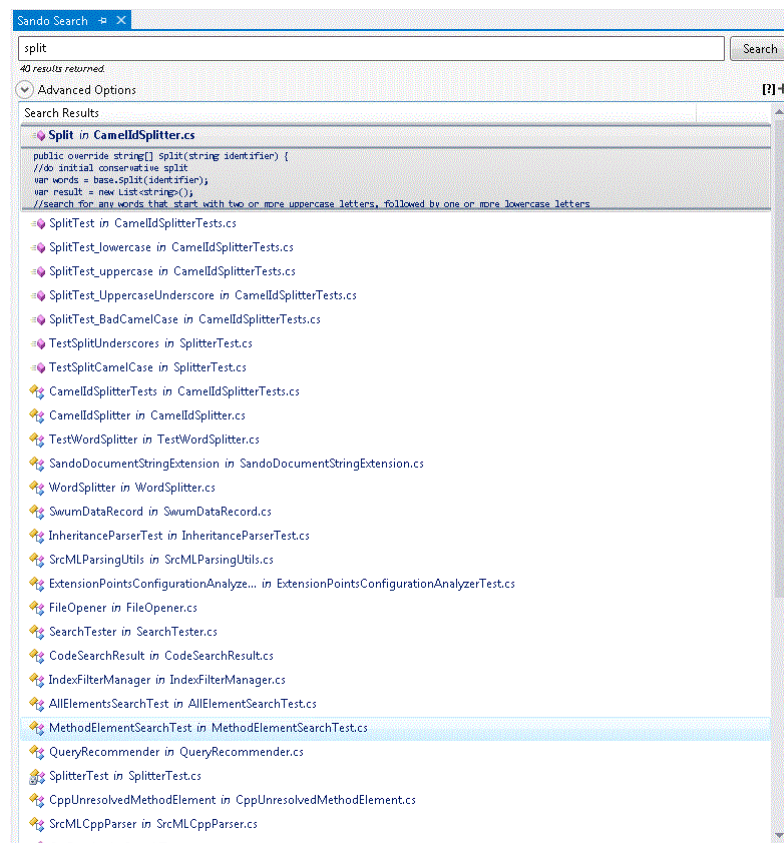


Figure 3.1: A screen shot from the tool Sando. The search term *split* is used to query the code base. For the search term *split* 40 recommendations of points in the source code are suggested. The first result is expanded, such that the relevant code snippet is visible. With a double-click on any result one can jump into the source code.

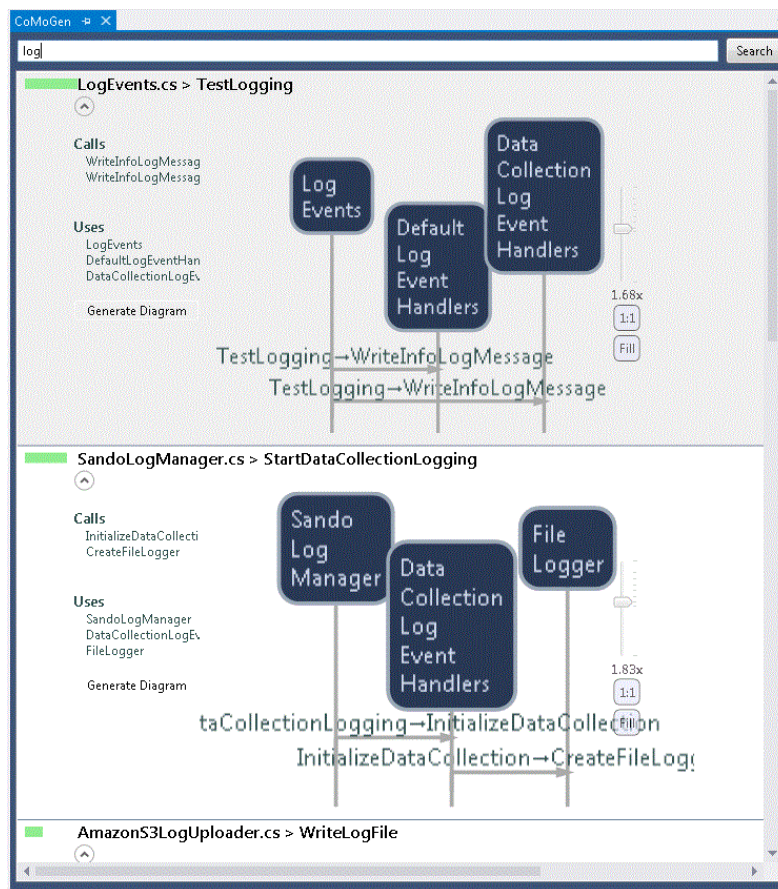


Figure 3.2: A screen shot from the tool Comogen. The search term *log* is used to query the code base. For the search term *log* 6 recommendations of points in the source are suggested. With a double-click on any method call one can jump into the source code.

Change Task	Summary	Description
1	Sando fails to properly handle underscores in query and corpus	Sando fails to match identifiers that contain an underscore, when the query is not quoted. Also, queries containing an underscore do not match terms in the index. Probably a splitting issue.
2	Sando isn't robust to Lucene directory removal	If the Lucene directory (in C:\Users \<username>\AppData \Local \Microsoft\VisualStudio \...) somehow gets removed, Sando seems confused at startup, looking for the cached Index there. On Sando startup, we need to make sure that the Lucene directory still exists before reading the Index.
3	Small, unuploaded log files should be removed	Small log files (< 400) bytes are not uploaded or deleted and will pile up at the user's machine. A better solution is to erase them once we decide that they are too small for log collection.
4	Searches on terms similar to access level specifications return junk	Searches for words similar to access level keywords collide with the access level declarations, which are stemmed before being indexed. For instance, a query term "protection" will match all protected methods. There is no reason for the access level keywords to be stemmed.
5	Subprojects in solution are not indexed	Sando does not index projects recursively, so if a project contains a subproject, the files within it will get ignored.
6	Parsing C++ needs to be performed via SrcML Service	The C++ parser still generates its own SrcML files. Using the SrcML service is the preferred way to do this.

Table 3.1: The six change tasks of the Sando project investigated by the ten participants.

points regarding the specific change task. The participants of each group rated the change tasks about equal in terms of difficulty.

3.1.5 Tools and Instrumentation

In this study, we used Visual Studio 2012 Ultimate running on a remote desktop with Windows 7. For each change task the according revision of the Sando source code was opened by the experimenter. The participants were allowed to use Sando and Comogen and additionally the common source code navigation features of Visual Studio 2012 Ultimate, such as *go to definition*, *view call hierarchy* and *find all references*. The experimenter only answered questions about fundamental unclarities within the change task which were caused of non-native English speaker participants. As example, the experimenter explained if needed the word "stem".

During the study execution we recorded audio, we captured the screen and monitored navigation steps within Visual Studio. The monitor within Visual Studio captured the commands *find in files*, *find in selection*, *find next*, *find prev*, *find regular expression*, *find whole word*, *go to definition*, *find all references*, *view call hierarchy*. Further all keyboard inputs were captured and the selections on methods, classes, events, properties, fields and files in the solution monitor. Additionally, the elapsed time per change task was measured and the participants answered a questionnaire about

their confidence about the points found in the code, their confidence about the query terms used and their perception of the difficulty of the change task. After the participants were finished with all change tasks we conducted a semi-structured interview to learn more about their experiences during the investigations of the change tasks.

3.1.6 Procedure

We executed the study with each participant separately in our lab. Additionally to the participant only the experimenter was present. First, the participant received a questionnaire to fill in his personal data. Then, each participant got a short introduction from the experimenter to the features of Sando and Comogen and was allowed to experiment with both tools for maximum five minutes. The participants received further instructions which were put down on paper (see appendix A). They were asked to identify for six change tasks three places in the source code (at least on method level) they would point a new programmer to, who will be assigned this task to work on. Further, they were asked to speak out loud at which elements they were looking at and what they were doing while investigating a change task. Also the timed limit of ten minutes was explained on the instruction note and the participants were pointed out to all recording mechanism of the study. Then, the experimenter answered specific questions about the procedure, such as which navigation features were allowed. The experimenter told the participant in which group they are and then they started investigating the change tasks. After finishing with each task, the participants filled out a questionnaire which captured the confidence about the found points in the source code, the confidence about the query terms entered in the search tools and the perception about the difficulty of the task. After finishing with all the change tasks, we conducted a semi-structured interview with each participant to learn more about their general experience with the two different tools.

3.2 Study Results

We transcribed the Sando and Comogen specific actions of 8.24 hours of video and audio capture together with the files produced by the Visual Studio monitor. The list of navigation steps in the file produced by the monitor was complemented with Sando and Comogen specific actions as the monitor was not able to capture events from an extension within Visual Studio. One part of an audio capture was corrupt and hence could not be analyzed. We could counteract this flaw, as we have screen recordings and files produced by the monitor which include the participant's actions on the change tasks. The audio capture is not corrupted during the follow up semi-structured interview. As we unexpectedly had to change the configurations within the Visual Studio environment, the monitor did not produce meaningful files. Only the keyboard inputs were correctly captured but not the specific developer actions. Hence, we built the transcription for the first three participants completely manually out of the screen and audio captures. We crosschecked the completely manually created transcriptions with the transcription created out of the correct monitor files complemented with Sando and Comogen specific actions we observed in the screen and audio captures. We did not notice any deviations in the two types of transcriptions.

We analyzed the following dependent variables:

- the number of navigation steps accomplished during an investigation of a change task
- the time elapsed until the participants finished with an investigation of a change task
- the relevancy of the found starting points
- the confidence of the found starting points

- the confidence of the entered search terms
- the number of queries accomplished during an investigation of a change task
- the use of the navigation features *View Call Hierarchy* and *Find All References*
- the number of results jumped in
- the number of queries accomplished without jumping in a result

For all dependent variables, we chose to do a Mann-Whitney test as we are considering a categorical treatment with different participants and data points which do not meet parametric assumptions³. The participants are different in each category because they did not work on a change task with both treatments. The Mann-Whitney test ranks all the datapoints from both treatment groups. If the the datapoints from both treatment groups are similar, then each group contains about the same number of low and high ranked datapoints. On the other hand, if the treatment groups differ, one group contains more lower ranked datapoints and one group contains more highly ranked datapoints. This difference is even more visible, when the ranks of the datapoints are summed up. To determine whether there is a significant difference between the datasets, the z-score is calculated using the mean and the standard error over all datapoints [Fie05]. Table 3.2 summarizes the findings of the statistical tests. We report for each dependent variable the median, the interquartile range, the test statistic U, the z-score, the significance and the effect size. We calculated the effect size on each statistical test to report a standard measure of the size of the effect we observed in our study, such that these results are comparable to observations from other studies. If the parametric assumptions were met on the datasets the counterpart test which can be applied when the parametric assumptions are met (independent t-test) would be more powerful, in terms that the independent t-test would more likely discover an effect if there is one in the dataset. However, since we are dealing with datasets which do not fulfill the parametric assumptions we cannot calculate the statistical power, since we do not know the Type I error rate [Fie05]. Further, we analyzed the answers to the questionnaires from the participants' investigations on the six change tasks. In the following our key observations with respect to the research questions are discussed.

3.2.1 Browsing Code Structures to Find Task Relevant Points

To understand if the presentation of a context along with the recommendation results supports the developers in finding interesting points in the source code, we analyze first the number of navigation steps accomplished. Through a Mann-Whitney Test we found that the number of navigation steps from Comogen users ($Mdn=18.5$) is significantly lower than the number of navigation steps from Sando users ($Mdn=23$), $U=312.5$, $z=-2.036$, $p=0.042$, $r=-0.263$. We assembled the number of navigation steps through the transcription of the files, which were produced by screen capture together with the Visual Studio monitor. While the number of navigation steps is lowered when using Comogen, the relevancy of the starting points found, the time needed to come up with starting points, the confidence about the starting points found and the use of the navigation features *View Call Hierarchy* and *Find All References* do not differ significantly (see table 3.2).

To determine the relevancy of the starting points found, we compare these points to the committed changes of the particular change task. We assigned an adjacency of zero to a starting point found which was recorded in the committed changes, an adjacency of one to a starting point found which is reachable within one navigation step from any committed change of the change

³according to the Kolmogorov-Smirnov test

dependent variable	median, range		U	z	p	r
	Sando	Comogen				
navigation steps accomplished	23,11.5	18.5,10	312.5	-2.036	0.042	-0.263
relevancy of the found starting points	4,6.75	1.5,4	365.5	-1.307	0.191	-0.169
time elapsed (in seconds)	600,178.25	572,158.25	420.0	-0.465	0.642	-0.060
number of queries accomplished	4,3.75	3,2	305.5	-2.163	0.031	-0.280
confidence of the found starting points	3,2	3.5,1	378.5	-1.090	0.276	-0.141
the confidence of the entered search terms	4,2	3.5,2.5	426.0	-0.365	0.715	-0.047
use of navigation features <i>View Call Hierarchy</i> and <i>Find All References</i>	1,2.5	1,2	388.5	-0.941	0.347	-0.121
number of jumped in results	4,4	2,3	291.0	-2.385	0.017	-0.308
number of queries accomplished without jumping in a result	1,1	1,2	332.5	-1.802	0.072	-0.233

Table 3.2: The results of the Mann-Whitney test on all the dependent variables. Comogen and Sando differ significantly in terms of navigation steps accomplished, number of queries accomplished and the number of jumped in results.

task. In this way we assigned an adjacency of two, respectively three to starting points found reachable within two, respectively three navigation steps. By considering the size and structure of the Sando source code and observing that only four out of 60 change task investigation instances included navigations sequences with more than three steps, we decided to account only starting points into the relevancy measure which are reachable within three navigation steps. The time was measured with a stop watch and the confidence was captured on a scale from 1-5 in the questionnaire each participant filled out after finishing with each change task. We assembled the number of invoking the navigation features *View Call Hierarchy* and *Find All References* through the transcription of the screen capture and the Visual Studio monitor files. Along these results we observed in a Mann-Whitney test that developers made significantly less searches when using Comogen ($Mdn=3$) than when using Sando ($Mdn=4$), $U=305.5$, $z=-2.163$, $p=0.031$, $r=-0.28$.

We hypothesize, that developers overall interact less with the system when using Comogen because the developers are able to assess search results better when a context is provided. The context prevents developers of jumping into not related parts of the system and spending time navigating irrelevant code. The higher number of queries accomplished in Sando supports this hypothesis as we observed that developer often jumped into source code recommended by top results in Sando, then they quickly scrolled through the class and entered another query term. On the other hand, developers using Comogen were not forced to jump into the search result to assess if it is meaningful. The presentation of the context did not cause a significantly decreasing number of the navigation features *View Call Hierarchy* and *Find All References* even though this kind of information may have already been assessed while choosing the recommendation result in Comogen. On average, the participants using Comogen accomplished 1.3 times such a feature and the participants using Sando accomplished on average 1.634 times such a feature. We did not report a significant difference on the time elapsed while the participants worked on the

change tasks. As the participants using Comogen made significantly less queries, we claim that the participants using Comogen assessed the recommendation results more accurately and spent less time browsing through source code than the participants using Sando. The participants who used Comogen found about even relevant starting points although significantly less navigation steps were accomplished. We claim that this improvement originates from the more accurate assessment of the recommendation results which display a context. Participants using Comogen and participants using Sando are about equal in terms of the confidence of the found starting points, even though participants using Comogen navigated less. We hypothesize that the comprehension of the source code which participants using Sando gained while navigating code, was already captured by the participants using Comogen while assessing the recommendation results.

Analyzing the transcription, created out of the screen recordings and the Visual Monitor files, we identified a specific sequence of navigation steps carried out by participants of both groups. The sequence of steps includes the jumping into a result, then going to a definition of an element and then going back immediately to the origin again. The last two steps of the sequence, following a definition of an element and returning to the original element, are arbitrarily repeated. We identified twelve appearances of this pattern, while nine appearances of the pattern are carried out by participants using Comogen and three appearances of the pattern are carried out by participants using Sando. We hypothesize that Comogen users jump into a result after they assessed the context of the recommendation and recognize terms in the source code which they have seen before in the context, but cannot remember what the connection was. Thus they navigate to these elements just to quickly see what it was about and then immediately return to the original result.

3.2.2 Browsing for Task Related Recommendation Results

Another key observation of this study is that developers using Sando more often missed relevant search results in the recommendations than developers using Comogen, even though Sando generally recommended a greater number of relevant results. We base the claim that developers missed more relevant results in a recommendation set of Sando on the observations from the screen and audio captures. As we analyzed the screen capture along with the audio contents, we wrote down each time a developer missed a relevant result in the recommendation list. Over all 60 change task investigation instances we observed 16 query instances in which a developer missed a relevant result in Sando and we observed one query instance in which a developer missed a relevant result in Comogen. We hypothesize three reasons for this behavior.

First, we hypothesize that the developers could better identify relevant search results when the context of a recommendation was presented. We base this hypothesis on the findings of the semi-structured interview (see appendix C for the quintessences of the interviews) and the dependent variable “queries accomplished without jumping in a result”. In the interview, nine participants stated that they used lexical information pieces to assess if a result was relevant. Sando displays, when no code snippet is extended, two terms per search result and Comogen on average around five⁴. The additional code snippet which can be extended in each Sando search result, contains also terms which are irrelevant to the query, such as “public static void”. Thus, given the broader range of displayed terms in Comogen, enabled the participants to choose relevant results. The number of queries accomplished without jumping into any result is about equal in both groups, while the participants using Sando missed more relevant results than the participants using Comogen. Thus, we claim that the context supports the accurate assessing of relevant recommendations.

Second, we hypothesize that the order of the search results is a highly influencing factor for choosing to jump into a result. Therefore, we determined the position within the result list of both tools

⁴This is a subjective observation from the experimenter

of the relevant search results, generated with the last query term entered. We chose to examine the last accomplished query of each change task because the developers generally queried in an advanced stage of the change task investigation with better search terms. Overall results of the last query, Sando includes 16 recommendations of points in the source code which are included in a change set of an examined change task and Comogen recommends overall six relevant points in the source code. The difference between these figures originates from the fact, that the participants made significantly more queries with Sando during an investigation of a change task (see table 3.2) and hence came up with better last search terms. However, we found that Sando presents the relevant search result on average as the 15.69th result (± 11.15), while relevant results in Comogen are presented on average as 3rd results (± 3.53) and are further within the top two result sets presented. To determine the location within the result set, we count the lines of results until we reach the relevant result for Sando. For Comogen, we count every method invocation until the relevant search result is reached. Thus, the developers using Comogen were not forced to scroll through the result set to find relevant results.

Third, we hypothesize that the bigger size of the result list in Sando (a maximum of 40 recommendations are displayed out of which 30 fit in a screen which is 1024 x 768 pixels) did not improve the selection of relevant recommendations. We base our hypothesis on the findings of the semi-structured interviews and the significantly higher number of accomplished queries together with the significantly higher number of results jumped in when using Sando. As Sando user accomplished more queries and jumped into more search results within about equal time span as Comogen users investigated change tasks (see 3.2), we infer that only top-ranked Sando results were investigated and thus relevant results further down in the list were missed. Furthermore, a participant stated in the semi-structured interview that he looked only at the first five results in Sando. This observation supports the results of Starke et al. [SLS09] which observed that developers usually look only at one search result and enter another search instead of browsing through the rest of the result set.

3.2.3 Finding Good Search Terms

We observed that one challenge participants face in the initial search phase, is coming up with good search terms. Even though Sando suggests, depending on the letters fed in by the developer, some related search terms, both tools were able to recommend only few points in the source code which were edited because of the change task. Both groups were also about equally confident about the search terms entered (see table 3.2). Table 3.3 sums up the number of all relevant results found with the first search term entered, the number of all relevant results found with the last search term entered, and the sum of all relevant results generated with search terms in between the first and the last query. We observed that in an advanced stage of the change task investigation, participants come up with better search terms, as the first search term entered by each participant on each task generated only nine relevant recommendations while the last search term entered generated 22. Over all participants and change tasks 118 queries were accomplished in between the first and the last query, which resulted in merely 18 relevant recommendations.

We hypothesize that too general search terms, as well as a too specific search terms were entered in the code search tool. Hence, too many irrelevant, respectively too few results were generated. Participants who entered too general search terms were overwhelmed by the size of the resulting list. In the interview a participant stated that if he has to browse through hundreds of results, he would just start on top and ignore the rest. Participants who entered too specific search terms used the wrong vocabulary. The usage of a wrong vocabulary originated probably from the programming experiences of the developers. We also hypothesize that combinations of terms were

Change Task	CT 1	CT 2	CT 3	CT 4	CT 5	CT 6
First Search Term	0	2	5	0	1	1
Middle Search Terms	2	8	2	1	1	4
Last Search Term	2	6	6	3	2	3

Table 3.3: The numbers of relevant recommendations generated by both tools of the first search terms, the numbers of all relevant recommendations generated through all search terms after the first input and before the last input and the numbers of relevant recommendations generated with the last search terms.

used where only one part of the query entered was leading to irrelevant result, preventing the good part of the query term to perform better.

3.2.4 Qualitative Analysis of the Semi-Structured Interview

This section summarizes the most important findings we gained through the answers of the participants in the semi-structured interviews. In this interview we asked the participants the following questions:

- What did you most like within Comogen/Sando?
- What did you like less within Comogen/Sando?
- Is there a particular strategy for selecting search terms?
- How did you decide which search results to explore further?

We analyzed the 72 minutes interview recordings by noting down the quintessence of each answer of each participant. Then we combined very similar answers into a general statement (see appendix C for the quintessences of the interview).

Six participants stated that they did not like the presentation of the context in Comogen, because the sequence diagram was too hard to understand, too much adjustments of the view have to be done prior to exploring or the diagrams were too big. One participant stated that he thinks it is a generally good idea to display context and two participants stated that the sequence diagram helped them to gain more confidence. Two participants explained that they would prefer a list or a tree view instead of a sequence diagram, which contains too much information. Two other participants suggested to present such a diagram optionally to the user. Another notion which was mentioned by six participants is that they were confused when looking through results which did not contain any related term to the concept and hence it was not comprehensible why a certain result is suggested. One participant stated "If I knew why a certain result is in the list, I would be more confident in assessing if its relevant or not". The participants had different opinions on the code snippets presented by Sando. Four participants found them useful to assess if the result is relevant and other four participant said they ignored the code snippet completely. Similarly the participants were split up in one group which likes to have test classes included in the result sets and in another group which were annoyed by the test classes in the search results. The common notion was that test classes should be marked somehow to differ them from other classes. Another suggestion on the improvement of both tools was to include also source code comments in the result set and a suggestion to the improvement of Comogen was to include any kind of outline over all results.

Regarding the strategy of the search term selections all participants explained that they tried to find terms out of the change task, which are very specific to this concept.

Nine participants assess the relevancy of a recommendation result depending on the terms they see. One participant even stated, that when assessing a search result in Comogen, he was looking at the additional terms and ignoring the information on the sequences of calls. This quote of a participant captures the general notion on how they decided to explore a search result further: "First, I assessed the class name and then the method name and if both matched the concept in my head, I selected the result". Several quotes of different participants are evidence to suggest that the location of a search result within the result set is influencing if a search result is selected: "In Sando, I looked only at the first 5 results", "In Sando, I oriented myself with the first search result, because scrolling the list is too tedious" or "If I have hundreds of results, I start on top and ignore at first hand the results further down the list". Another participants stated that he explored the results in Comogen more accurately and again another participant stated that he had more confidence in the recommendation of Comogen, because less results were presented.

3.3 Threats to Validity

In the following, we discuss several points which threaten the validity on the results of this exploratory study.

3.3.1 External Validity

Due to the small number of participants and the rather ordinary size and complexity of Sando's source code our results are limited with regards to generalizability. The participants had different levels of programming experience, which influenced the rapidity of understanding unfamiliar source code and which influenced the selections of search terms. The fact that the participants used the code search tools for the first time and investigated completely unfamiliar code is also somewhat artificial, but not avoidable as we conducted the study in our lab. The size and complexity of Sando is not comparable with a system which includes hundreds of thousands of lines of source code. The size of the source code may affect the size of the result sets of Comogen and Sando which are investigated by the developers. The change tasks which were investigated in our study could be categorized as "quick fixes" as these change tasks caused the editing of on average 3 methods (± 1.291). Further, our study is based on a program which is written in C# and the study was conducted within Visual Studio 2012 Ultimate. These two factors limit the number of reasonable navigations in the source code to object-oriented programming languages and the tools available within Visual Studio 2012 Ultimate.

Further, several constraints were imposed on the behaviors of the participant while investigating a change task. As example, the participants were not allowed to consult the internet or to discuss with other developers. Finally we imposed a time constraint of ten minutes.

3.3.2 Internal Validity

In the following we analyze several possible threats with regards to the internal validity of this experimental study. First, the time the participants spent on investigating a change task may be influenced by other factors than only by our independent variable. As example, learning effects throughout the six change task can occur, as the participant learn more about the system as they investigate more change tasks. We counteract this learning effect by applying a counterbalance measure design. Further, also the exhaustion of the participants of solving several change tasks in a row can influence the time measurements, as well as the individual perceptions of the task difficulty. The presence of the experimenter may also be an influencing factor on the time measurement, as the participants may feel uncomfortable being observed. The learning effect may

also influence the number of accomplished navigation steps, as the participants get to know the system better and thus can better assess if it is meaningful to explore, as example, a method call further. Further, the number of jumps into search results could be influenced by the prototypical status of Comogen. In Comogen, when only one method in a result set is recommended, it is not possible to jump directly in the code, because of missing mouse click listeners. The participants have to manually open the recommended point. This prototypical behavior of Comogen may have influenced the participants to jump into fewer results than they would have done in a fully developed tool. Other small bugs and prototypical characteristics of Comogen may have also influenced the dependent variables we investigated. As example, if a name of a class or a method is longer than expected by Comogen, it is cut off and the user cannot see the full name. Further when reselecting the Comogen tab within Visual Studio the mouse was in a mode which caused the moving of the prior selected diagram. Clicking on the diagram stopped the move mode again. The scrolling within Comogen was somewhat confusing, as it is not smooth and jumps over results. Further, the search took longer than the searches in Sando.

3.3.3 Construct Validity

We observed participants using Comogen to accomplish significantly less searches than participants which used Sando. We hypothesize that participants can distinguish relevant from irrelevant results better when the context of the recommendation is also presented and thus do not need to accomplish many searches until they find an interesting point. An alternative explanation for accomplishing more searches within Sando and less searches within Comogen are the automatic search terms recommendations of Sando. Participants may feel encouraged to try out the other search terms which are suggested by Sando. On the other side, in Comogen, the participants have to come up with search terms without external help, which can lead to less manifold ideas for search terms and hence to less queries. Further, the number of accomplished navigation steps is somewhat dependent on the number of accomplished queries and hence also influenced by the automatic search term suggestions of Sando.

3.3.4 Statistical Validity

We report as a possible measurement error the time measurements as the participant were told to press the stopwatch when they are finished before 10 minutes elapsed. Some participants forgot to press the stopwatch and had to be reminded to do so. Therefore, the time measurements on each change task could change slightly. Another measurement error is the perception of relevant points in the source code. We regarded changed methods in the according change sets to be the most relevant points in the source code for a change task. But a change set includes only the points in the source code which were edited and not points which account for the understanding of the change task. Therefore, the participants may have suggested points in the source code to look at, which we regarded irrelevant, but would have been assessed to be relevant if we would also have had the possibility to account the work during the accomplishing of a change task. Further, we defined that we account a starting point found in the relevancy measure if a changed method is reachable within three navigation steps. We defined the limit of three navigation steps, because of the size of the Sando project and because we observed in our study out of 60 change task investigations only 4 investigations which included more than 3 navigation steps from an interesting point in the code. The size of the project is insofar influencing as in a smaller project different concepts in the code can be reached within fewer navigation steps than in a bigger project. All of the results rely on the transcript of the developers' actions created out of the screen capture. Only one person made the transcription which was not crosschecked by other people.

The statistical validity is strengthened by the facts, that we used equal numbers of participants

in both groups, did not violate parametric assumptions when applying statistical tests which required to meet the parametric assumptions and we employed a counterbalance measure design to counteract learning effects.

3.4 Conclusion on the Exploratory Study

This exploratory study investigated the initial search and navigation phase before an actual change to the source code is made. The study included ten participants working on six different change tasks. Half of the change tasks were investigated by using a code search tool, which presents along with the recommendation results a context and the other half of the change tasks were investigated with a code search tool, which presents the recommendation in a list. We observed that one of the biggest challenges in the initial search and navigation phase is to come up with search terms, which provoke relevant results. Entering low performing search terms can originate from different causes. Either the developer enters a too general search term or a too specific search term or in case of a search term combination one search term can prevent the other to provoke good results. Further, we observed through the qualitative analysis of the semi-structured interview and the dependent variables “number of navigation steps accomplished during an investigation of a change task”, “time elapsed until the participants finished with an investigation of a change task”, “relevancy of the found starting points”, “number of queries accomplished during an investigation of a change task”, “the number of jumped in results” and “number of queries accomplished without jumping into a result”, that the presentation of a context along with the recommendation results supports developers in the initial search and navigation phase.

3.5 Implications from the Explorative Study

In this section we discuss the implications of the exploratory study we conducted with ten participants working on six change tasks. The study focused on the initial search and navigation phase before a change is made. We identified that one big challenge for developers is to find good search terms. Further, we found that presenting a context of a search result is supporting the developer in the initial search and navigation phase before editing the source code. We suggest a tool which automatically generates search terms for a change task at hand and we infer design implication for search tools which present a context.

3.5.1 Browsing Code Structures to Find Relevant Points

Our analysis of the explorative study revealed that developers using Comogen used about equally frequent the Visual Studio navigation features *View Call Hierarchy* and *Find All References*, even though Comogen users may already have assessed such type of structural information in the result set. We hypothesized that Comogen users jump into a result after they assessed the context of the recommendation and recognize terms in the source code which they have seen before in the context, but cannot remember what the connection was. Thus they navigate to these elements just to quickly see what it was about and then immediately return to the original result. To avoid this additional navigation steps, we suggest to have an additional small view of the most important pieces of the search result available in the code base when requested. As example, a keyboard combination could provoke a pop up or hovering over the suggested line of code could display a miniature context.

In the follow-up interview we found that most developers assess in a first step the terms contained within an element when navigating code in order to decide if the element should be explored

further. If the code element contains sufficient appealing terms, the detailed code structure is assessed as a second step. We suggest to display in a small view terms of the element, which are related to the initial query term when hovering over an element. In this way, a developer does not have to navigate to the element to assess in a first step its relevancy. If the terms of the element are appealing, a developer may be more confident that the element is relevant and spending time navigating the element is may be more focused as one can directly concentrate on the structure.

3.5.2 Browsing for Task Related Recommendation Results

We observed in the exploratory study that displaying context of a search result supports the developers in the initial search and navigation phase. There is evidence that developers can assess search results more accurately when a context is displayed, as participants using Comogen did not miss as many results as participants using Sando did. However, several participants disliked the sequence diagram of Comogen. The most mentioned flaw was the complexity of the diagram. Thus, we suggest to present the search result as a list which can be opened, when requested, to display a diagram containing the context. Further, we learned that participants assess the relevancy of a result mainly based on the terms which are related to the concept of the change task. On these grounds, we suggest to focus more on the terms which resembles to the query input, instead displaying specific calls. The presentation of the context would include the relationships of element which are displayed as a list of terms related to the query term (see figure 3.3 for a sketch of this possible design). We hypothesize that in this way developers would better understand why a certain search result is generated.

Further, we imply from the study results, that the number of search results has to be rather small and presenting the most relevant search results topmost. Overall, Sando and Comogen did not generate many relevant results. In total, 238 queries were accomplished during the study and merely 49 queries provoked a relevant search result. These rare strikes are on the one hand provoked by the selected search terms and on the other hand generated by the tools. The relevancy of the search result is only determined through the occurrence in the change set of the change task. Thus, the adjacency of search results to change sets is not taken into consideration within these figures. However, to generate more relevant results, we investigate whether taking into account user interaction histories can support developers in finding a starting context for change tasks (see section 4).

3.5.3 Finding Good Search Terms

We observed in our study, that one of the biggest challenges in the initial search and navigation phase, is to find appropriate search terms. The rather unsuccessful search terms entered by the participants, inspired us to prototype a tool, which includes two approaches to automatically recommend possible search terms given a new change task (see section 5). The search term suggestions generated by our tool can be used as input to a code search tool. Another suggestion to overcome this challenge, is prototyped in our approach of identifying a starting context for a change task at hand (see section 6). As this approach generates the recommendations on starting points fully automatically, given the change task, the challenge is avoided as the user does not have to think about query terms at all.

Another tool implication resulting from the observation of the entered search terms, is the possibility of localizing expert developers of a specific concept. The number of queries in each change task, which resulted in relevant recommendations can be analyzed. Out of the totally 60 investigation instances on change tasks, the queries of 27 investigation instances resulted in at least one relevant result. Out of these 27 investigation instances, 14 investigations are composed of queries

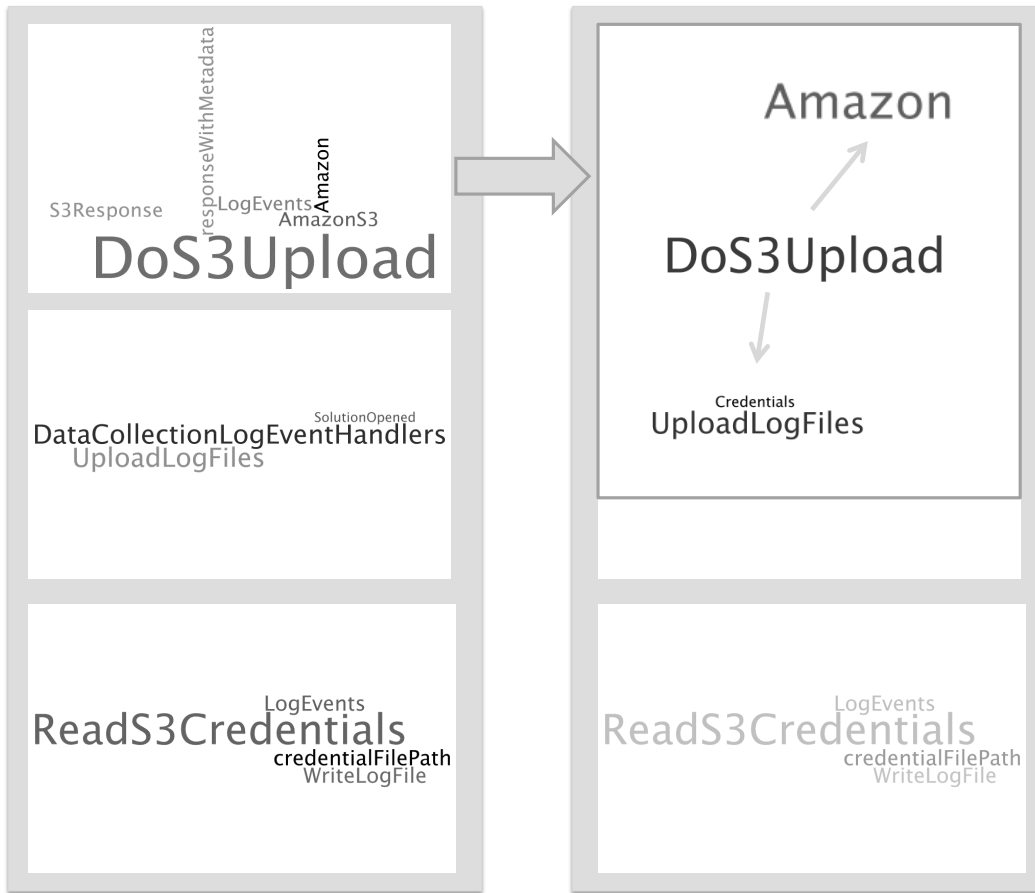


Figure 3.3: Sketch of a possible representation of the recommendation results along with a context.

out of which at least 50% generated relevant results. As example, participant 9 accomplished seven queries to investigate change task 2. Four search term he chose, resulted in at least one relevant recommendation. Participant 6 accomplished during the investigation of change task 1 seven queries and none of these resulted in a relevant recommendation. Therefore, an expert on a concept implemented by the change task, could be identified by analyzing the performance of search terms for a given task of each participant.

Using Interaction Histories to Recommend a Starting Context of Code Elements for a Change Task

Implications from our exploratory study we conducted to investigate the initial search and navigation phase before editing the code, include that search result lists, generated by code search tools, have to be rather small and have to present the most relevant results topmost. The state-of-the-art tools Sando and Comogen did not generate many results. From totally 238 queries conducted in the exploratory study, 49 queries provoked the presentation of a relevant result. This low number of strikes is explained on the one hand by the selected search terms and on the other hand this low number of strikes is generated by the tools.

Thus, we investigate whether we can support developers in finding a starting context for a change task by taking into account user interaction histories. We consider a typical static information retrieval approach for recommending interesting points in the source code for a new change task. Given a change task description, these approaches first identify similar change tasks that have already been completed. Completed change tasks have code changes associated to them. It is assumed that similar change tasks also changed similar code. The code changes of a similar change task can be recommended as starting points in the code for a change task at hand. In our research, we want to understand if we can enhance the retrieval of similar change tasks to a change task at hand by using developer interaction histories. Further, we investigate if taking into account user interaction histories yields good starting points for a change task at hand.

The following investigations are analyzing the project `org.eclipse.mylyn.context`. We chose to analyze this project, as it offers a broad number of change tasks with according change sets and user interaction histories. The user interactions are monitored in form of Mylyn task contexts. The project `org.eclipse.mylyn.context` implements the usage monitoring, the degree-of-interest modeling and the task-focused user extension for the user interface of the Eclipse plug-in Mylyn. The project `org.eclipse.mylyn.context` reports 209 change tasks, which are tracked in the bug tracking system Bugzilla¹. Further, the project comprises 2892 change sets in a git repository. Out of these 209 change task, 36 change tasks have a task context attached and are linked to a change set and are thus eligible for our analysis.

¹<http://www.bugzilla.org/>

4.1 Finding Similar Change Tasks

In this section we explore whether we can identify more precisely change tasks covering similar concepts to a change task at hand by using a task context. We investigate the situation for a completely new change task and we investigate the situation for a change task which has already been worked on, but is reopened.

4.1.1 Finding Similar Change Tasks to a New Change Task

When a new change task is submitted only few attributes are available which can be used to find similar change tasks. These features are summarized in table 4.2. Note that the features *comment*, *status*, *target milestone*, *assigned to*, *qa contact*, *url*, *whiteboard*, *keywords*, *depends on*, *blocks*, *cc list* and *see also* are not included in this comparison, as these features are not included in a new reported change task. We also do not take the attachments into consideration for this analysis, because the only attachment which is interesting for this purpose are patches, which are typically not filed along with a new change task. We identified two distinct ways of using a task context to find similar change tasks. We conducted a small case study for each of these two options.

Case Study 1

In a typical feature selection procedure, similar objects are determined and a pattern in the features of these objects is searched. This pattern of features is used to match change tasks to a change task at hand. In this case study we aim to determine a pattern of features once by using an initial set of change tasks with an overlapping task context and once with an initial set of change tasks which have common elements in their change sets. We compare which pattern of features found yields the better precision and recall in retrieving change tasks which are covering similar concepts Figure 4.1 depicts an exemplary course of this process when change tasks with a similar task context are used as initial set. The same process applies to the process where the initial set is determined through similarities in the change sets.

Study Method. In a first step, we determine all change tasks which share an element on method level in their task context, respectively which have a method in common in their change sets. Then, we determine manually which of these change task pairs are covering similar concepts. To find the best combination of features, we build all possible combinations and match each combination to the change tasks. We calculate the precision and recall of retrieving the change tasks covering similar concepts out of the change tasks which have similar task contexts, respectively change sets. The combination of features which retrieves the highest F-measure change tasks covering similar concepts is regarded as the best pattern.

To match a possible pattern to a change task we distinguish between exact matching and partial matching features. The features *tag*, *product*, *reporter*, *component*, *version*, *severity*, *hardware*, and *os* can be compared by exact matching while the textual features (*summary* and *description*) are compared in terms of their textual similarities. Based on preceding tests, in which different information retrieval models were employed, we use the vector model for this analysis and calculate the cosine between two document vectors to determine their similarity. To improve the accuracy of this textual analysis, we weight the index terms according the TF/IDF weighting method and apply in a defined sequence several text preprocessing steps. Due to prior lexical analysis, we process code snippets and stack traces, paths, punctuation, hyphens, upper and lower case, numbers, hyperlinks, names and abbreviations within change task particularly. Of course, we also stem the index terms of change tasks, and eliminate the stop words. In prior tests in which we

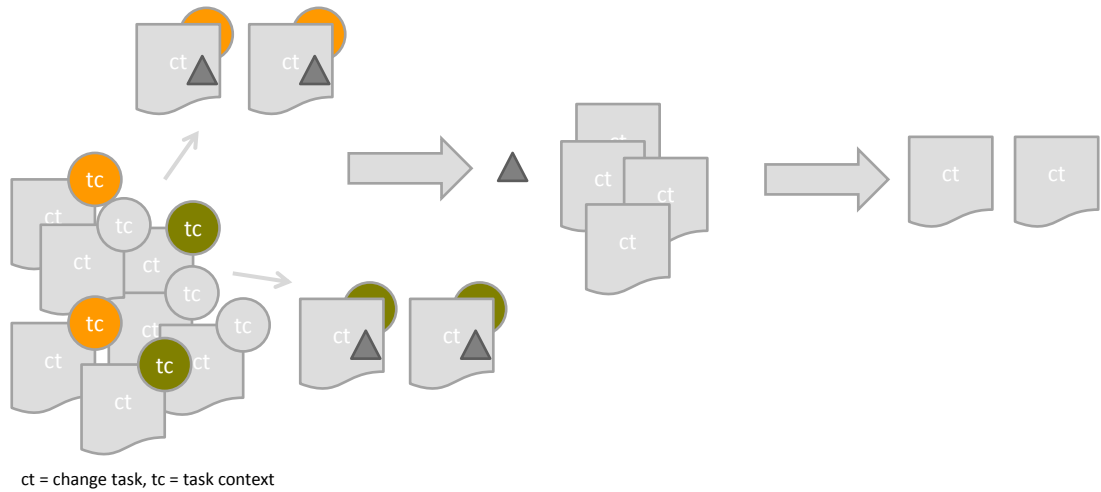


Figure 4.1: We identify the features which match in change task with a similar task context. Then we apply this pattern to a set of change tasks to determine change tasks which cover a similar concept.

used six different options of index term selections, we found that index term selection 1 is most appropriate for this corpus. These six index term selection options are summarized in table 4.3 and are based on findings of [BYRN99] and [BCCN94] (see appendix D for details on the text preprocessing). To determine which change task pairs are regarded textually similar and which are not a similarity threshold needs to be defined. This means that all combinations of change tasks pairs which similarity value is above the defined threshold are regarded to be similar and all combinations of change tasks pairs which similarity value is below the defined threshold are not regarded to be similar. Prior textual analysis showed that a threshold of 0.14 is appropriate for this corpus.

To understand if a pattern which originates from an initial set of change tasks with similar task contexts matches more precisely change tasks which are covering a similar concept, we used each of these eight patterns to match 36 change tasks from the project org.eclipse.mylyn.context. In this analysis we included change tasks which have a task context attached and are linked to at least one change set of the project. Then, we determined manually for each matched change task pair if the change tasks are covering similar concepts.

Results. We found several patterns of features for each initial set. These feature combinations are summarized in table 4.1. Table 4.1 also reports on the true positive and false positive retrievals we found through matching the identified patterns to change tasks of the project org.eclipse.mylyn.context. The feature patterns tc3, cs2 and cs4 are equally precise in retrieving change tasks covering similar concepts (with a precision of 0.667). On these grounds, we conclude that we cannot report any improvement on the preciseness of retrieving similar change tasks out of the project org.eclipse.mylyn.context using task contexts. We hypothesize that these two options can be used interchangeably.

Further, we also noted that the change task pairs determined through similar task context are substantially different to the change task pairs which we found through matching the change sets. We found eleven change task pairs with similar task contexts and ten change task pairs with

initial set based on	#	feature pattern	true positives	false positives
task contexts	tc1	os, summary, description	5	7
	tc2	tag, os, summary, description	1	4
	tc3	reporter, os, summary, description	4	2
	tc4	tag, reporter, os, summary, description	1	1
change sets	cs1	hw, summary, description	12	12
	cs2	reporter, hw, summary, description	4	2
	cs3	os, hw, summary, description	5	7
	cs4	reporter, os, hw, summary, description	4	2

Table 4.1: The combination of features for both initial sets together with the number of matched change tasks (true positives and false positives regarding the concepts covered by the pairs) out of 36 change task from the project `org.eclipse.mylyn.context`.

similar change sets. Only four pairs are in common. Rastkar and Murphy [RM09] made a similar observation, when they investigated if change tasks with similar task contexts include more pairs covering the same concept than change task pairs with similar change sets. The hypothesis is stated that task contexts and change sets capture rather different types of information. As example, the contextual information is missing in the change set.

Case Study 2

In this case study we compare the textual features of a new change task (*summary* and *description*) to the task contexts available for the project. In this way we aim to identify change tasks which cover a similar topic as the change task at hand. We investigate if matching natural language to task context elements yields to more precise retrieval of change tasks covering a similar concept.

Study Method. We match the natural language features of the change task at hand to an interesting element of a task context as described in section 3.5.3. We sum up the element matchings. The change tasks with the highest number of element matchings are suggested to be similar to the change task at hand. To find out if this approach performs more precise than the approaches in case study 1, we picked randomly seven change tasks out of the 36 analyzed change tasks of the project `org.eclipse.mylyn.context`. We manually assess the suggestions for each of the seven change tasks and calculate the precision.

Results. For each of these seven change tasks our approach retrieved one change task out of the change task for the project. We assessed three out of these seven suggestions to be true positives (precision of 0.429). We report that we cannot suggest more precisely change tasks covering similar concepts when matching text features of the change task at hand to task contexts available for the project.

²the detailed descriptions are taken from <https://bugs.eclipse.org/bugs/page.cgi?id=fields.html>

Change Task Feature	Description
summary	The summary is a short sentence which succinctly describes what the change task is about.
tag	A custom unknown type field in this installation of Bugzilla [†] .
description	The description of the change task.
product	A classification of the change tasks.
reporter	The person who filed this change task.
component	Components are second-level categories; each belongs to a particular product.
version	The version field defines the version of the software the change task was found in.
severity	How severe the change task is, or whether it's an enhancement.
hardware	The hardware platform the changed task is designated.
os	The operating system the change task is designated.
attachment	A content which is attached to a change task.

[†]Mylyn uses tags in the summary of the change task to categorize bugs. All tags used in Mylyn change tasks can be found here:
http://wiki.eclipse.org/Mylyn_Contributor_Reference#Bugzilla.

Table 4.2: The features of a change task which are captured when a new change task is filed ².

Index Selection Options	Description
option 1	Only nouns are selected for indexing and other grammatical groups are not considered.
option 2	Nearby nouns are clustered and these noun groups are used as distinct index terms.
option 3	Each term which is surrounded by whitespaces is selected.
option 4	Index terms from option 1, coupled with code related terms included in the text features.
option 5	Index terms from option 2, coupled with code related terms included in the text features.
option 6	Index terms from option 3, coupled with code related terms included in the text features.

Table 4.3: We identified six different index term selection strategies which are used to calculate similarity measures between text features of change tasks.

4.1.2 Finding Similar Change Tasks for a Re-opened Change Task

For a large software system, a large amount of change tasks are reported. Sometimes, a developer starts working on a change task and before completing the change she must switch to another task [KM06]. For such situations the developer created already parts of the final task context. Those bits and pieces of beginning task contexts can be used to find change tasks covering similar concepts. We conducted a small case study to find out if we can find more precisely change tasks covering similar concepts.

Study Method. We compare task contexts based on their interesting elements. We only consider elements which are on method level. Each element's *structure handle* of one task context is compared to each element's *structure handle* of another task context. The *structure handle* states the method declaration. We randomly chose seven change tasks out of the 36 analyzed change task of the project `org.eclipse.mylyn.context` and assessed if the suggested similar change tasks are covering similar concepts.

Results. We compared the task context of seven randomly chosen change tasks to each other task context from the project. The matching on the task contexts generated twelve change task pairs. We assessed seven out of these twelve change task pairs to agree on the concept they cover (precision of 0.583). This case study showed that, for the 36 analyzed change tasks, we cannot improve the preciseness on the retrieving of change task pairs which cover similar concepts.

4.2 Recommending Potentially Relevant Starting Points

In this section, we want to investigate whether a task context includes information, which can be used to recommend a starting point for a given change task. We hypothesize that using a task context for recommending starting points is convenient for several reasons. First, if we act on the assumption that similar change tasks also changed similar code, it is even more likely that similar change tasks *navigated* through same code elements. A task context captures not only the code elements which were edited because of the change task, but also the code elements which were navigated through. Further, task contexts educe a broad range of additional information, which, as example, is not captured within the information of the change set. As example, a task context contains elements which can be exactly weighted according their relevance to the change task at hand. The most important source code elements of a change task are not necessarily the code elements which were edited. Thus, when taking into account only the information of change sets, possibly important source code elements are disregarded. Additionally, the chain of user interactions can be precisely reproduced, which allows to conclude on the rationale for a specific code editing. One can also infer additional information from the change set, as example when taking into account the comments of a commit, the main reason of the code changes can be learned. However, the process of extracting such additional information out of change sets is tedious and not as precise as if a task context was used.

Because of these reasons we believe that using a task context for recommending starting points of a change task can support a developer in the initial search and navigation phase when performing a change task. To investigate this hypothesis, we elaborated different suggestions on how a task context can be used to recommend a starting point for a change task. These suggestions include the generation of a recommended starting point for a change task and the generation of a context

to search results.

4.2.1 Generation of Starting Points

This section assumes that a similar change task to the change task at hand is already identified (the identification of a similar change task is discussed in section 4.1). Following a typical static information retrieval approach for recommending interesting points in the source code, which assumes that similar change tasks changed similar code elements, we assume that similar change tasks *navigated* through similar code elements. We even hypothesize that similar change tasks have a stronger agreement on navigation sequences than on actual code changes, as code changes are very specific to a particular change task. On the other hand we assume that a specific concept is implemented within a partitioned module of the source code which is navigated when working on a change task which covers that concept. Hence, the challenge of finding starting point recommendations for a change task at hand, which are generated using a task context, is divided into selecting the most relevant elements from the task context (figure 4.3 depicts this concept). We identified two distinct approaches to extract relevant information from the task context of a similar change task. The first approach takes into account the degree-of-interest model [KM05a] and the second approach analyzes textual similarities.

Using the Degree-of-Interest Model of Mylyn

The Eclipse plug-in Mylyn monitors all interactions of a developer while working on a change task. The captured source code elements which constitute the developer's context of the change task are encoded within a degree-of-interest (DOI) model. To bring out the relevant elements of a change task, the Mylyn user interface decodes the DOI model when a task-focused filter is enabled. The DOI model captures the relevance of each element to their change tasks by associating an interest value to each element. A set of default scaling values, which were adjusted upon usage statistics, is responsible for assigning adequate interest values to the source code elements. As example, a selection of a code element increases its value by 1, a typing on the keyboard by 0.1 while the interest values of the other code elements decrease by 0.1. The DOI model not only captures the selections and edits of the developer, but also so-called propagation events. As example, when an element within the same package is multiple times selected or edited, the package also gets an interest value assigned. Recency is also an influential factor when assigning interest values to a code element. When a code element is not selected nor edited over a specified amount of time its interest value decreases. When an interest value decreases under a specified threshold then the element is removed from the DOI model. The threshold which defines if an element is interesting is equal 0³. The developer working on a change task can also manually remove uninteresting elements from the task context or manually add interesting elements to the task context of the change task [KM05a].

This approach of generating starting point recommendations uses the DOI model to elicit the most interesting elements of a change task similar to the change task at hand.

The change tasks of the project `org.eclipse.mylyn.context` include on average 54.7 (± 38.33) interesting elements in their task contexts. This number of interesting elements is too high for taking them as recommendations, as we observed in our exploratory study that a starting point recommendation list supports developers best, when it is short and precise. Between all change tasks and elements of their task context the DOI interest value is on average 96.37 (± 341.76). Within each change task the DOI interest value has a standard deviation of on average ± 116.2 . Thus,

³We refer to the implementation of `InteractionContextScaling` in `org.eclipse.mylyn.internal.context.core` when change `I4bfa906f137eb7dd3cb5be67c50cd2460806e0a9` was committed.

we cannot specify a threshold to identify the interesting elements for the starting point recommendations. Hence, we analyzed the task context elements which interest values are within the top three interest values of the particular change task and we analyzed the task context elements which interest values are within the third quartile of the interest values of a particular change task. This analysis included 36 change tasks of the project `org.eclipse.mylyn.context`. We found, when taking into consideration the top three elements, that for ten change tasks, the recommended starting points will result in elements, which were mainly edited. The examination of the elements within the third quartile revealed, that for nine change tasks, the recommendation will result in elements, which were mainly edited. For these change tasks, our approach will result in similar starting point recommendations as an approach which uses change sets. Since, the sum over all interest values of top three task context elements is 12.69 time higher than the sum of the task context element of the third quartile, we chose to recommend the task context elements which have a top three interest value associated.

Analyzing textual similarities

This approach is based on two assumptions. The first assumption includes that the task context of a similar change task includes relevant elements for a change task at hand and the second assumption includes that terms used in the textual features of the change task are reflected in the class and method declarations of the source code. In this approach we are matching the terms included in the textual features of a change task at hand to the terms included in each element of a similar change task.

We preprocess the natural language terms as described in appendix D to better match them to source code terms of the task context elements. We extract the class name and the method name from the task context elements and lowercase the terms. Then we assess which set of task context element terms includes the most matches with the set of terms from the textual features of the change task at hand.

This approach is somewhat similar to the approach we suggested for finding search terms (see section 5.1). The main difference is that the search term recommendation approach treats the words included in the source code term corpus in a so called bag-of-word representation while this approach treats the terms of each change task element separately.

To get a general idea of how these two approaches perform in recommending starting points for a change task at hand, we chose to generate starting points for six change tasks of the analyzed project using the implemented prototypes. As both approaches rely on the prior assessment of a similar change task, we manually entered a similar change task for each one assessed. We compared the generated starting points of each approach with the particular change set of the change task. We defined a starting point recommendation to be relevant, if the specific element was also represented in the change set. The approach which used the degree-of-interest model generated for four analyzed change tasks starting point recommendations, which are relevant, while the text matching approach generated for two change tasks relevant results. As the task context element terms usually consist of solely four different terms, we hypothesize that the text matching approach needs a bigger corpus to perform better. Thus, we conclude that the text matching, we applied between the textual features of the change task at hand and the task context elements, is not suitable for finding relevant task context elements.

We also found, that the *relevant* search results generated by the degree-of-interest model are based on task context elements, which mainly constitute of selection user interaction events. Only one relevant result out of six relevant results overall, was mainly based on editing user interaction events. This finding supports our hypothesis that similar change tasks *navigate* through similar

code elements and less edit similar code elements.

4.2.2 Generation of a Context

We hypothesize that a task context is also convenient for generating a context for a search result, which was generated by a code search tool. Different to other context generation approaches, this approach does not rely on structural information within the source code. We observed in the exploratory study (see section sub:BrowsingCodeStructures) that developers often conduct a specific sequence of navigation steps when navigating through source code. Thus, we analyzed the sequences within the interactions of the user while performing a change task. Specifically, we looked into the interactions immediately before a specific interaction and the interactions immediately after a specific interaction (see figure 4.2). We observed mainly two patterns. The first pattern includes the interaction elements $e1$, $e2$, $e3$. At time $t1$, the developer visits $e1$, then navigates to $e2$ and then visits $e3$. At time $t2$, the developer visits $e3$, then navigates to $e2$ and goes back to $e1$. The second pattern includes the interaction elements $e1$ and $e2$. A developer starts the investigation at $e1$, navigates to $e2$ and returns to $e1$.

The second user interaction pattern strengthens the observation of the user behaviors we made in the exploratory study. We hypothesize, that given a relevant starting point recommendation which may be generated by a code search tool, task contexts enable the generation of a completely connected context graph. It is subject to future work to find more user interaction patterns within a task context, which can be used to generate a connected context of code elements for a change task automatically.

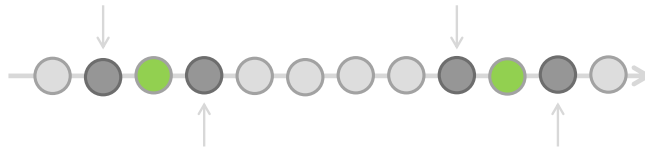


Figure 4.2: To investigate if user interaction patterns are also reflected in the task contexts, we analyzed the interactions immediately before a particular interaction and the interactions immediately after.

4.3 Conclusion

We investigated on the basis of case studies on the project `org.eclipse.mylyn.context` whether we can improve the preciseness of finding similar change tasks and whether we can recommend starting points using a task context. We report that using task contexts, when identifying similar change tasks to a change task at hand, yields equally precise change task suggestions. We suggested a pattern based on task context information, which is able to match similar change tasks. Only very few change tasks of one specific project were analyzed, which limits the generalizability of these case studies. Additionally, the change tasks analyzing in the case study on re-opened change tasks, included closed change tasks, which limits the significance firmly. This is due the fact, that the change task included in the analyzed group of change tasks of the project `org.eclipse.mylyn.context` contain only two change tasks which *status* equals new.

We showed that when task contexts are considered to recommend starting points for a change task

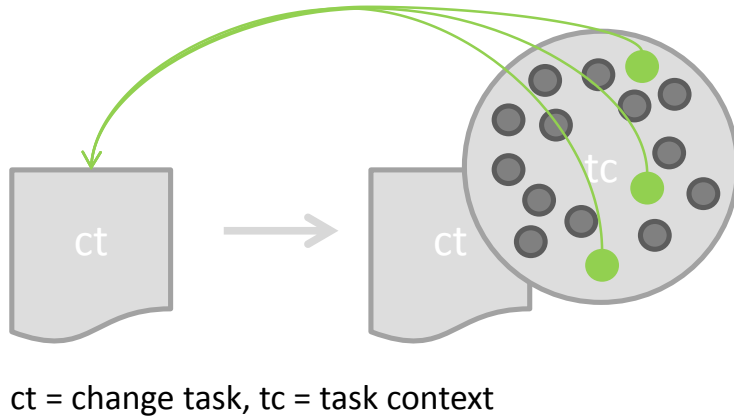


Figure 4.3: Our approach generates recommendation for starting point based on task context elements of a similar change task.

at hand, the recommendations are in 66.67% of the cases relevant. Further, the investigation on the task contexts of six change tasks strengthened our hypothesis, that the generated recommendations are mainly the *navigations* which two change tasks have in common and not necessarily the edited parts.

Through analyzing single user interactions within task contexts, we found two specific patterns. These patterns can be used to complement a recommendation of a code search tool. In future work, we plan to automatically generate a connected task context for a given change task. Task contexts include a variety of meta-information pieces which could be exploited to support developers while performing a change task. As example, the total amount of time a developer spent within one specific element could be examined. We hypothesize that depending on this time span, the complexity of the element could be inferred. Further, we hypothesize that elements which are included in many task contexts are a hint for possible bottlenecks in the source code. The importance of a change task and hence the importance of the according source code elements, could be inferred by analyzing task switches and time spans. Additionally, we hypothesize that the values generated by the degree of interest model also reflect, as example the excitement of the developer, as we observed in our study, some developers start to select more within a smaller time span when working on a difficult task, which in turn would influence the DOI. Nevertheless, the task context with the according degree-of-interest values always represent a specific filter for a specific point in time. Meaning that, as example, the elements in the code which were navigated to explore an unknown API, are initially important and eventually disappear from the task context. However, for a new change tasks, these elements would have been probably a good starting point.

Two prototypes for Automatically Recommend Search Terms for a Change Task

This chapter describes two prototypes of a tool which automatically recommends search terms for a given change task. These tools are a direct result from the observations of the exploratory study we conducted with ten participants to investigate the initial search and navigation phase before a change is made (see section 3). In our exploratory study the participants used for half of the change tasks investigated the code search tool Sando¹. As a user types in characters into the search field, Sando suggests search terms. These suggestions heavily depend on the typed in characters of the user. As a result of our study, we report that none of the participants using Sando entered for change task 1 and for change task 4 a first query term, which generated a relevant result in the recommendation list. For the other change tasks the participants entered first search terms, which generated results on average at the 13.71th (± 11.7) position in the result list of Sando. These findings let us conclude, that the suggestions of Sando are not necessarily supporting developers, which are unfamiliar with the system, in finding immediately good search terms. To overcome this challenge, we suggest two approaches of automatically recommending search terms given the textual features of a change task.

To automatically recommend search terms for a given change task, we use the task contexts which are attached to priorly completed change tasks of the system. We hypothesize that using a task context to recommend search terms has two advantages. Namely, the corpus of all possible search terms is tremendously decreased as against when all terms (i.e. class names, method names and variable names) of the source code are processed. Nevertheless, using task contexts enables to suggest source code vocabulary close-by terms. Thus, a code search tool which receives our recommended terms as input, does not need to map them to the vocabulary used in the source code base.

5.1 Approach 1 for Recommending Search Terms

In order to generate search terms for a given change task, we first determine similar change tasks, as we assume that similar change tasks cover similar concepts. This work discusses the identifi-

¹<http://sando.codeplex.com/>

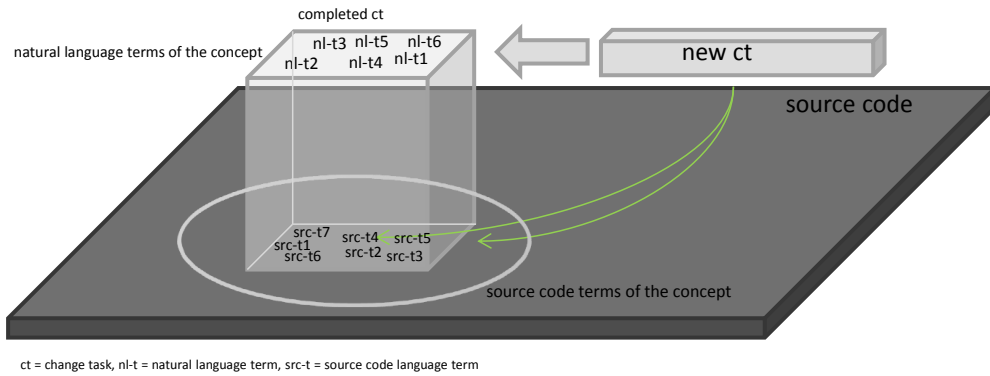


Figure 5.1: Our first approach to automatically recommend search terms includes the identification of a similar change task. Within the term corpus on source code level of the concept, we identify terms which describe the new change task best.

cation of similar change tasks in section 4.1. Further, we assume that the elements in the source code are implemented using a vocabulary which reflects the particular concept too. The elements which were navigated around while working on a change task are reflected in the task context of the change task. These elements represent the corpus of the terms which is identifying a particular concept with the vocabulary of the source code. Thus, we conclude that a concept of a change task can be either identified with the vocabulary of the textual features within the change task or with the vocabulary of the code which was navigated when the change task was performed. We also assume that these two sets of vocabularies have overlapping terms, as element names in the source code are chosen with the aim to identify its behavior. Hence, the challenge of finding good search terms for a given change task can be divided into finding good terms in the source code corpus of the particular concept. Figure 5.1 depicts the concept of our first approach.

To find good search terms for the change task at hand, we match the textual features of it to the identified terms included in the source code term corpus of the concept. In this way, we tremendously scale down the possible search space to the concept of the change task, which is expressed in source code vocabulary. As the task context captures every interaction event of the user while working on a change task, we only consider the *elements* of the task context, which are compiled by Mylyn. Further, we consider only elements which are on method level, as the remaining elements are too general as to identify a concept precisely. To better match terms of different vocabularies (i.e. natural language to source code), we accomplish some preprocessing activities. The natural language tokens are built by splitting the sentences on each whitespace and on each minus sign. We split the terms explicitly on the minus signs too, as many terms which describe an action are composed terms (e.g. on-hover). These action specific terms are more likely than other terms to appear also in the source code vocabulary. Further, we remove English stopwords and remaining punctuation as these terms are not likely to appear in the source code vocabulary. We do not stem the terms, as too much information for our matching strategy would be lost. For each element in the task context, we extract the class name and the method name. These terms are split according to the camel case rule, as actions are often described using more than one word. As example, the source code element `addPage` is split into `add` and `Page`, which both are likely to appear in natural language vocabularies.

To find the elements which most likely contain a good search term for the change task at hand, we match each set of element terms to the set of natural language terms of the change task. We count how many times a term of the element set contains a term from the natural language set and how many time a term from the natural language set contains a term from the element set. The element sets with the highest number of containments are assumed to identify a subset of the concept of the change task expressed in source code vocabulary. We extract the terms of the highest matching element sets which are responsible for the high matching. Our approach suggests these terms as search term recommendations for the given change task.

5.2 Approach 2 for Recommending Search Terms

Our first approach (see 5.1) has one main limitation. Namely the finding of similar change tasks to the change task at hand. If change tasks covering different concepts are regarded to be similar to the change task at hand, our approach will most likely not find any search terms. To overcome this flaw, we implemented a prototype of a second approach which does not rely on specific other change tasks of the project.

As we explicated in section 5.1 we assume that a particular concept can be either expressed with natural language terms or can be expressed with source code terms. We also assume that concepts can have overlapping terms in both corpora. As we assume that a specific concept expressed in the terms of the natural language corpus is reflected in the terms of the source code corpus, we conclude that one specific term of the natural language corpus points to one or more specific terms in the source code corpus.

This approach aims to map the natural language terms which are used overall change tasks to the terms which are used overall in the source code. To find adequate search terms for a change task at hand, our approach looks up the natural language terms used for the change task at hand and suggests the source code terms which are associated to them. As change tasks of the same software system generally use an alike vocabulary, it is likely that the natural language terms of the change task at hand are covered in the mapping. As soon as the change task at hand is completed the mapping is updated. Hence, the challenge of finding a good search term for a given change task can be divided into switching between these layers of used vocabularies. To better understand the approach figure 5.2 illustrates the concept. As example, a new change task includes in its natural language text features the terms *nl-t17*, *nl-t18* and *nl-t19* (see figure 5.2). Our approach looks up the mappings to the source code terms and retrieves for *nl-t18* the source code terms *src-t16* and *src-t12*. For *nl-t17* the source code term *src-t21* is retrieved and for *nl-t19* no key in the mapping exists.

To create the mapping of the natural language terms to the source code terms, our prototype retrieves all change tasks of the project including the associated task contexts. We regard the entries in the task contexts as the elements which contain the terms which constitute the corpus of the source code terms. Although our approach would also work with other kinds of information sources, which identify source code elements, such as a change sets. As in the first approach (see 5.1) we only use the *elements* of a task context. As we aim to suggest only a few, but precise terms, we only take into consideration the elements which describe an source code element on method level. If we would consider also elements which are on a higher level, the corpus of the source code terms would explode and too many source code terms would be matched to a single natural language term. The textual features of the retrieved change tasks (i.e. the *summary* and the *description*) constitute the terms of the natural language corpus. Both types of terms undergo a preprocessing workflow, which enables to better match terms within the same type of corpus. For the natural language terms, we apply the preprocessing steps which are explained in appendix D.

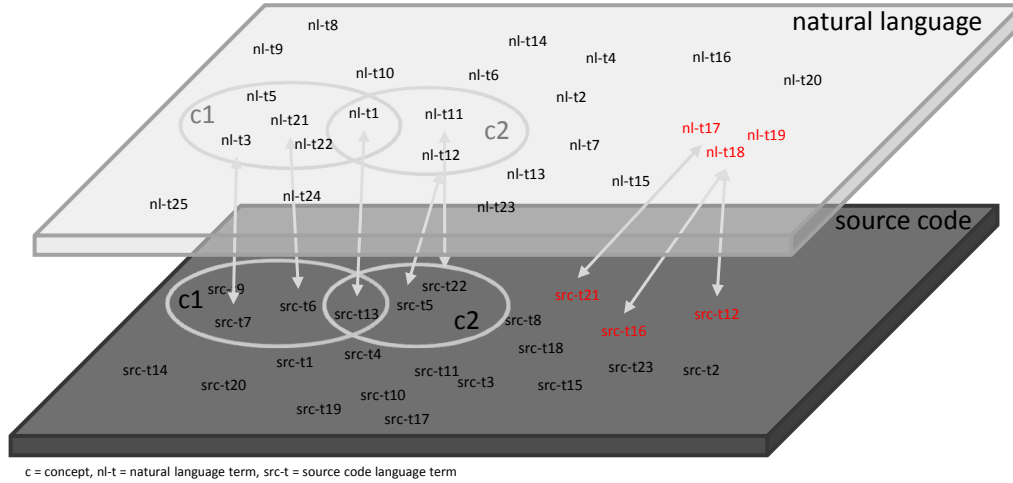


Figure 5.2: The concept of our second approach. We assume that terms of the natural language term corpus match one or more terms of the source code term corpus. A set of natural language terms defines a concept of a change task, which can be mapped to a set of terms in the source code term corpus. As example, *nl-t1*, *nl-t11* and *nl-t12* define the concept *c2*, which is expressed as *src-t13*, *src-t5* and *src-t22* in the source code.

In this procedure the terms are stemmed, meaning that the term *filtered* and *filtering* are processed to *filter*. After the preprocessing steps on the terms of each change task, a change task is represented as a set of stemmed words. For the task context elements, we retrieve the class name and the method name of each element, split them according to camel case and remove any additional punctuation. Further, we setup a small collection of stopwords which threat the usability of this approach. The stopwords collection includes terms, such as *can* or *get*, which typically appear in an object-oriented source code. These terms are excluded from further analysis. After the pre-processings on the elements of each task contexts, we found for each change task the terms on the natural language level and the terms on the source code level. To get the mapping of one specific natural language term to one or more specific source code terms, we assign firstly to each natural language term of a change task all the source code terms of the particular change task. We count the frequencies of the source code terms which appear in each same term of the change tasks. This means, that if a change task 1 includes the natural language terms 1,2 and 3 and a change task 2 includes the natural language terms 3,4 and 5, we assess which source code terms are in common between the terms attached to term 3 of change task 1 and the terms attached to term 3 of change task 2. The frequencies of the source code term attachments in common for each same term overall change tasks is the decisive factor for the mapping.

For a new change task, the natural language text features are preprocessed in the same way as for the completed change tasks. Our approach looks up the natural language terms of the new change task and return the according source code terms to the user as suggestions for search terms.

5.3 Prototypical Implementation

We implemented a prototype of our approaches to automatically recommend search terms as an Eclipse plug-in. The user of our prototype has to provide the textual features of the change task

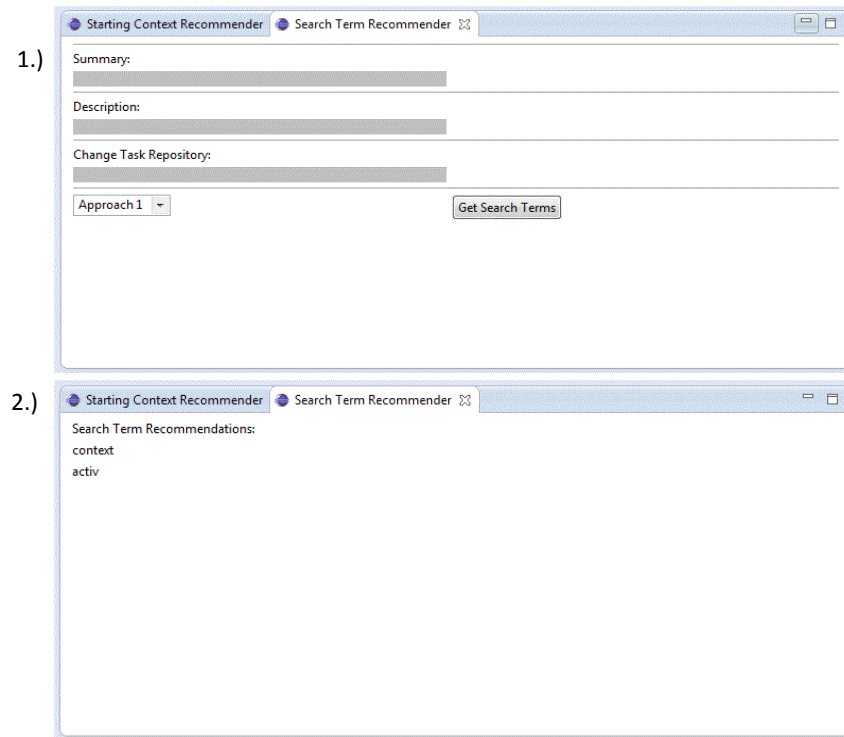


Figure 5.3: The search term recommender views. 1.) is a screenshot of the user entries and 2.) is a screenshot of the result view which includes the suggested search terms.

of which she wants to get search term suggestions and a url to the change task repository, which stores the completed change tasks of the project. Further, the user can select which approach she wants to use. See 1.) figure 5.3 for a screen shot of this view. When clicking the button *Get Search Terms* our prototype starts to retrieve change tasks and the according task contexts from the specified url. Our prototype implements both approaches which we discussed in section 5.1 and 5.2 to recommend search terms depending on the use selection. The recommendations according to the approach selected, are presented to the user as depicted in 2.) of figure 5.3.

5.4 Case Study

To understand if search term suggestions generated by our approaches are good, we chose to conduct a small case study on four change tasks of the project `org.eclipse.mylyn.context` (see table 5.1). We determine if a search term is good, if its usage in a code search tool results in at least one relevant result. A relevant result is determined through the change set of the change task being assessed. If a result generated by the code search tool includes a method which is included in the change set of the change task, we define that the search term entered was good.

For the first approach, we determined similar change task manually to get an idea on how the approach performs, if this limitations is dispatched. We used the Eclipse File Search tool to query the 376 files of the project `org.eclipse.mylyn.context` with the generated search terms. We used the search terms as part of a regular expression and search only within files ending with `".java"`. We queried the code base of the project with different combinations of the suggested search terms.

As example, for change task 175655 we queried the code base with the combinations `[.*hover.*]`, `[.*listen.*]` and `[.*hover.*listen.*|.*listen.*hover.*]`. In most cases, it was very obvious which term combinations are most promising.

The first approach generated on average 2.75 (± 1.48) suggestions of search terms and the second approach generated on average 10 (± 4.53) suggestions. (see table 5.1 for the analyzed change tasks and the generated search terms). Table 5.2 summarizes the findings on relevant results, generated through the Eclipse File Search tool, we retrieved through the best combination of the suggested terms. We counted only results pointing to different methods as distinct results in the Eclipse File Search tool result list.

5.5 Discussion

We applied our approaches only to a small number of change tasks, which limits the generalizability. However, including the task context into the recommendation of search terms looks promising. Even though, approach 1 suggested for one of the change tasks five different search terms and approach 2 suggested for two change tasks 14 search terms. Although, we think that building good combinations of suggested terms, given the change task text, is not challenging. Both approaches suffer limitations. Approach 1 is dependent on the identification of a change task which covers a similar concept and approach 2 is not usable when the set of terms from the textual features of the change task is too big. Approach 2 suggests a probably exponentially growing number of search terms depending on the size of the input set. Thus, we think, that approach 2 is more suitable when a query is entered by the user, which is likely not as long as the textual features of the change tasks. Approach 2 is essentially a translation engine which transform a user query in natural language into source code language which is more adequate for a code search tool.

We hypothesize that we found good results, because we could exclude efficiently irrelevant terms from possible suggestions. Both approaches are not dependent on user input, which may be an advantage when developer which are unfamiliar with the system start a query.

Future work can be done on excluding terms within both corpora which would yield to many search results. Thus, an extended stopword list for the specific source code vocabulary is needed. Our approach can be used in any code search tool, which requires the user to enter a query, as an initial feature to support developers in finding a good search term. Task contexts are not always available for a software project. Interchangeably, elements extracted from change sets could be used instead. However, we hypothesize that elements extracted from change sets do not provide as good search terms as when a task context is employed, because change sets only include a subset of the vocabulary, which is identifying the concept of the change task.

Change Task	Summary	Description	Suggested Search Terms 1	Suggested Search Terms 2
175655	[api] [context] provide an on-hover affordance to supplement Alt+click navigation	many times I want to get a set of files from a package into the context. currently the process is pretty annoying, for each file I need to: * alt-click the parent package. * click on the file. since the alt-mode turns off when I click the file, I need to do the whole thing again. it gets extra annoying if I am in hierarchal package presentation, and I need to 'drill down' the packages list for each file. maybe a solution can be a new 'sticky' alt click mode, that remains until I explicitly turn it off? (ALT-CTRL maybe?)	hover, listen	unfiltered, mouse, decoration, children, hover, drawer, landmarks, browse, focused, listener, bridge, viewer, move, filters
364155	[api] add a method to retrieve projects in current active context	Build Identifier: Review is here: http://review.mylyn.org/#change,111 Adds a method to retrieve all IProject, that are contained in the currently active context. Reproducible: Always	ui, project, activ	projects, landmarks, editor_se_3_8_2
340622	alt+click hover affordance appears when no task active	I am seeing the alt+click hover affordance in the package explorer without a task active. It went away after closing and reopening the package explorer, but when I restarted Eclipse, it was there again.	hover	mouse, filtered, decoration, children, hover, drawer, landmarks, browse, listener, focused, filters, viewer, move, preferences
267143	[performance] [context] expand all causes expensive computations in InterestFilter	When all nodes in a view are expanded the interest filter can cause a noticeable slow down of the UI.	interest, select, viewer, filter, up	unfiltered, handle, providers, focused, listener, interaction, tasks, elements, filters

Table 5.1: The four change tasks of the project org.eclipse.mylyn.context for which we automatically generate search terms.

Change Task	Result Length 1	List	Relevant Results Approach 1	Result Length 2	List	Relevant Results Approach 2
175655	5		5	5		5
364155	3		1	1		0
340622	5		0	6		2
267143	15		8	3		0

Table 5.2: The number of results generated by the Eclipse File Search tool using the best combination of the suggested search terms and the number of results which point to methods which were changed within a change set which is linked to the change task.

Approach: Automatic Identification of a Starting Context

We suggest an approach to automatically identify starting points for code exploration given a change task at hand. We base this approach on the observations from the explorative study (see section 3) and the findings from the case studies conducted on the change tasks and task contexts of the project `org.eclipse.mylyn.context` (see section 4). We developed a prototypical implementation of our approach, which can be used in future work to examine its feasibility. The prototype is implemented as an Eclipse plug-in.

6.1 Usage Scenario

This section illustrates how a software developer can use our approach for finding starting points in the source code for a change task at hand.

The software developer John has joined the Mylyn software development team recently. He already worked on the Mylyn software system, but is not familiar with the source code at all. Today, he gets a new change task assigned. He understands that the change task is about some unexpected behaviors in the package explorer when hovered over, but he has no clue where to jump into the source code to understand more about the problem domain.

So, John opens in Eclipse the view *Starting Context Recommender* and enters all required information, such as the change task id, the url to the repository containing the other change tasks of the project and finally he enters the path to his local git repository (see figure 6.2). Then he clicks on *Get Recommendations*. A result view is presented (see figure 6.3) which includes the recommendations for starting points of John's change task. John immediately strikes the recommendation `MouseListener` and `FilteredChildrenDecorationDrawer`. He jumps into the recommendation `MouseListener` by double-clicking on the result. He notices that `MouseListener` is a private class of `FilteredChildrenDecorationDrawer` and that it is related to mouse hovering, since he finds the method `mouseHover` in this class. At this point John is indecisive what to do, because `mouseHover` seems to work appropriately. So, he decides to go back to the *Starting Context Recommender* view to explore more results. He notices that the third result in the recommendation list includes the terms `applyToTreeView`, `addListener` and `addMouseListener` and hence he jumps into the method `applyToTreeView` where he finally finds the missing if-statement which caused the change task.

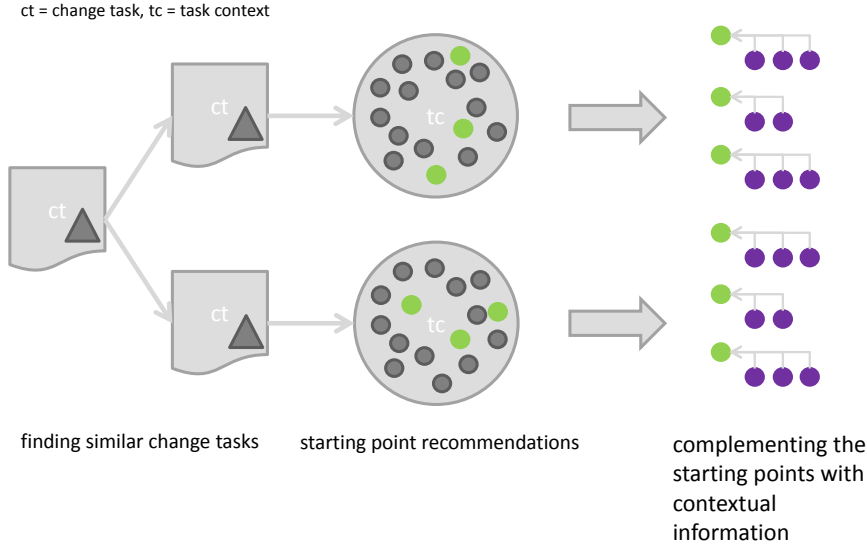
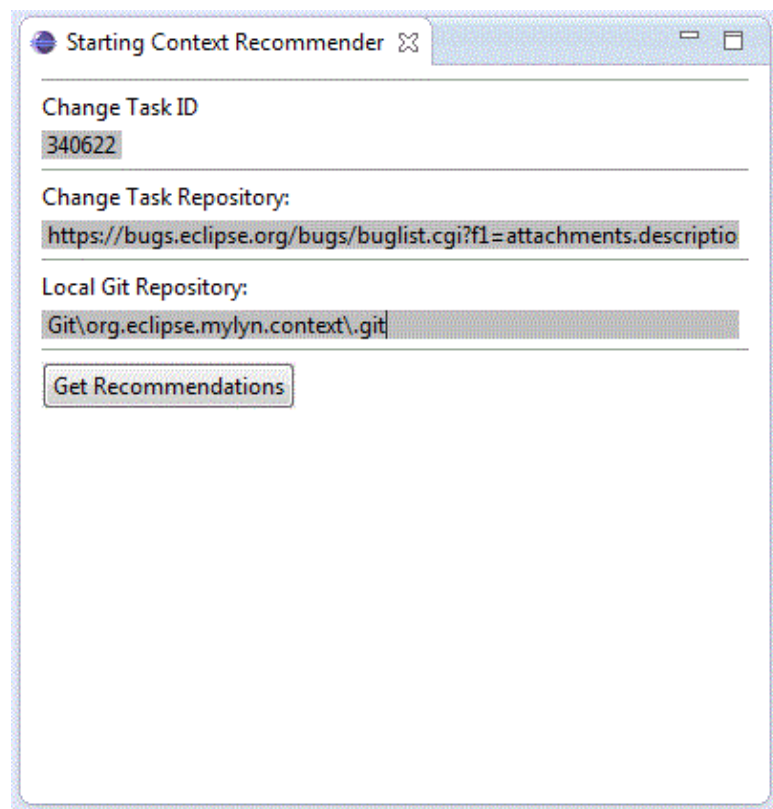


Figure 6.1: The workflow of our approach to automatically recommend a starting context for a change task at hand. First, similar change task are determined through a feature pattern. Then, using the degree-of-interest model, relevant elements of a task context are pitched on. These elements are the recommendations for potentially relevant starting points for the change task at hand. Finally, we complement the recommendations with structural information of the source code.

6.2 Identification of a Starting Context

To recommend a starting context for a change task at hand, our approach comprises three distinct parts. First we retrieve out of the set of reported change tasks all similar change tasks to the change task at hand. Then, we recommend the top three task context elements according to the degree-of-interest model as starting points for the change task at hand. In a third step, these starting points are complemented with contextual information, which we infer from the source code structure of the project.

Our approach is based on the assumption that the feature pattern *tc3*, which is specified in section 4.1, table 4.1, matches similar change tasks to a change task at hand. Thus, a change task is retrieved, and regarded as similar, if it has the same values for the features *reporter* and *os* and if the cosine similarity of its textual features is at least 0.14. To determine relevant elements from the task contexts of the similar change tasks, we use the degree-of-interest model. The determined elements of the task contexts are recommended as initial starting points for the change task at hand. To provide a context for each recommendation, we look up the method invocations of the relevant element in the source code structure. At this stage, our prototypical implementation cannot include the methods which invoke the relevant element, as this procedure consumes too much computation time. Figure 6.1 depicts the workflow of our approach.



The screenshot shows a window titled "Starting Context Recommender". It contains three input fields and a button. The first field, labeled "Change Task ID", contains the text "340622". The second field, labeled "Change Task Repository:", contains the URL "https://bugs.eclipse.org/bugs/buglist.cgi?f1=attachments.descriptio". The third field, labeled "Local Git Repository:", contains the path "Git\org.eclipse.myllyn.context\.git". Below these fields is a button labeled "Get Recommendations".

Figure 6.2: To use our tool, the user has to provide the id of the change task to investigate, the url to the change task repository and the path to the local git repository. If all information is provided, the user can click the button *Get Recommendations* to generate suggestions for an initial starting context. In this screenshot one queries an initial starting context for change task 340622.



Figure 6.3: The recommended starting points for change task 340622. One can jump into the recommendations by double-clicking the result.

Implementation Details

This section describes the prototypical implementation of our approach to automatically identify a starting context for a change task at hand. We implemented our approach as an Eclipse plug-in. Our implementation integrates with the change task repository Bugzilla. Although our approach is extensible to other change task repositories.

The implementation consists of mainly six components: View, Starting Context Recommender, Data Model, Similarity Engine, Connector and Source Code Parser (see figure 7.1). In the following, we characterize each component in more detail. Furthermore, we briefly disclose implementation details of the search term recommender 3.5.3 and the Visual Studio Monitor which we used in the exploratory study 3.

7.1 Data Model

Our implementation is based on an underlying model. This model consists of the change task model, which reflects several features of a Bugzilla change task. The model captures the *id*, *summary*, *description*, *product*, *reporter*, *component*, *version*, *severity*, *hardware* and *os* features of a change task. Further, a comment model and an attachment model are included in the change task model. Each of these models capture relevant pieces of information regarding the comments, respectively the attachments. The change task model includes the information to load the priorly downloaded zip file which includes the task context. The task context is stored in an xml file. We hook the `LocalContextStore` of `org.eclipse.mylyn.internal.context.core` to work with the task context.

7.2 Starting Context Recommender

This component defines the workflow of our approach. It receives the user input and downloads the change tasks. Then it queries the similar change task to the specified one by invoking the similarity engine. Then it retrieves the interesting elements of each similar change task's context by hooking the `IInteractionContext` of `org.eclipse.mylyn.context.core`. The relevant elements are fed into the source code parser which determines the callees of the relevant elements. Finally, the starting context recommender weights the terms according to their occurrences and passes the information to the view, which displays the result to the user.

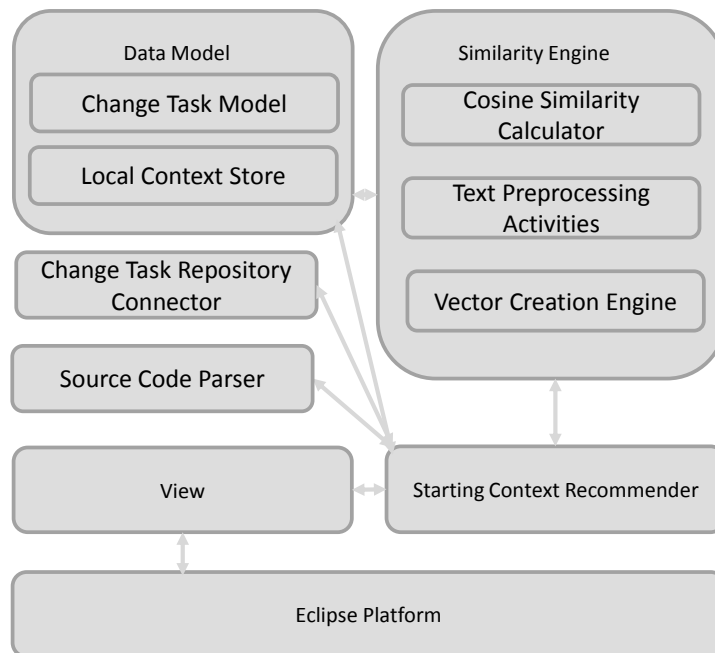


Figure 7.1: Overview of the five main components included in the architecture of our prototypical implementation.

7.3 View

So far, our implementation provides two views, implemented with SWT, to interact with a user: the entry view and the results view. In the entry view, the user can provide the details needed for our approach, such as the change task id, the url for the Bugzilla tracking system and the path to the local git repository. The results view displays the task context. The results view responds to double-click events on a result to open the specific file in the Eclipse editor. See examples of these view in figure 6.2, respectively figure 6.3.

7.4 Connector

Our implementation includes a connector to the change task tracking system Bugzilla. Although the approach works with any change task tracking system. To connect to the Bugzilla change task repository, we hook the *BugzillaRepositoryConnector* of the project `org.eclipse.myllyn.internal.bugzilla.core`. The user of our prototypical implementation only has to provide a query. One can query the analyzed set of change tasks in the browser application of Bugzilla ¹ and then copy and paste the url into our prototypical implementation. The connector includes an adapter which adapts the queried change tasks from Bugzilla to our change task model. The adapter also downloads the change tasks' task context zip files and stores them locally. Furthermore, we also implemented a git repository connector which was employed during the case study 1 of section 4.1 to retrieve change sets of change tasks.

7.5 Source Code Parser

To complete a recommended starting point automatically with additional context, we analyze the call dependencies of the particular project. To do so, we read all *.java* files of the project and build an abstract syntax tree by using the `org.eclipse.jdt.core.dom.ASTParser`. Through a visitor pattern we search the abstract syntax tree for the method, which is recommended as starting point. Then we look up all method invocations of the found method declaration. We also implemented the mechanics to find the callers of the recommended method. This procedure includes searching in each method declaration of a particular method invocation. We could not include this piece of functionality in our prototypical implementation as the computation consumes too much time.

7.6 Similarity Engine

The similarity engine includes all functionality required to match two change tasks. Specifically, it preprocesses the textual features, creates vectors out of the preprocessed textual features and calculates the cosine similarity between two vectors. The specific preprocessing activities are described in Appendix D. At this point we complement the report on text preprocessing activities. We report on the code snippet detection mechanism, which is part of the preprocessing activities. We assume that the analyzed project is written in Java and the code conforms to the Java coding conventions ². Inspired by the work of Nicholas Sawadsky [Saw12], we define a regular expression which determines if text is regarded as Java code. Specifically, the regular expression is constructed through the following rules:

¹<https://bugs.eclipse.org/bugs/query.cgi>

²<http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367>

- the term starts with one or more lower-case letters AND contains at least one upper-case letter; OR [Saw12]
- the term starts with an upper-case letter AND contains at least one lower-case letter, followed by an upper-case letter; OR [Saw12]
- the term starts with two or more upper-case letters AND contains at least one lower-case letters [Saw12].
- the term starts with either one or more lower-case or upper-case letters AND contains a dot which is followed by at least one lower-case letter.

To create vectors out of the preprocessed textual features, we used the class `Matrix` of the Jama project³. These vectors are weighted according to the TF/IDF (term-frequency-inverse document frequency) weighting mechanism. We calculated the cosine between these vectors using the formula [SB88]:

$$\text{similarity}(Q, D) = \frac{\sum_{k=1}^t w_{qk} \cdot w_{dk}}{\sqrt{\sum_{k=1}^t (w_{qk})^2 \cdot \sum_{k=1}^t (w_{dk})^2}}$$

In the above formula, Q denotes the query vector, D denotes the document vector, w_{qk} denotes the weight of the term k of Q and w_{dk} denotes the weight of term k of D .

7.7 Search Term Recommender

The search term recommender is part of the Eclipse plugin we built for our prototypical implementation of our approach to automatically recommend a starting context. Both views can be opened in Eclipse through *Window → Show View → Other → Starting Context Recommender tools* (see 7.2). The search term recommender uses `BugzillaConnector 7.4` to retrieve the change tasks of the project specified in the url which is provided by the user. The search term recommender hooks the `LocalContextStore` of `org.eclipse.mylyn.internal.context.core` to parse the task contexts into `IInteractionElements`. We implemented a text preprocessor for the text features used in the search recommender. This text preprocessor includes a subset of the preprocessing steps which are described in appendix D. Particularly, we do not stem words in this case, so that they can better be matched to the terms included in the structure handles of the `IInteractionElements`. The preprocessing steps on the task context elements are specified in the search term recommender.

7.8 Visual Studio Monitor

To monitor the navigation steps, the keyboard inputs and the selections conducted by the participants within Visual Studio, we implemented a Visual Studio extension. Several navigation steps are hooked by a command interceptor. This visual studio extension can handle events from the following commands: *clean solution, find, go to definition, open class view, clear breakpoints, close all documents, close document, close solution, open debug explorer, enable breakpoint, F1 help, file close, file open, find backwards, find in files, find in selection, find match case, find next, find prev, find all references, find whole word, go to error tag, insert breakpoint, open output window, paste, pause, open project*

³<http://math.nist.gov/javanumerics/jama/>

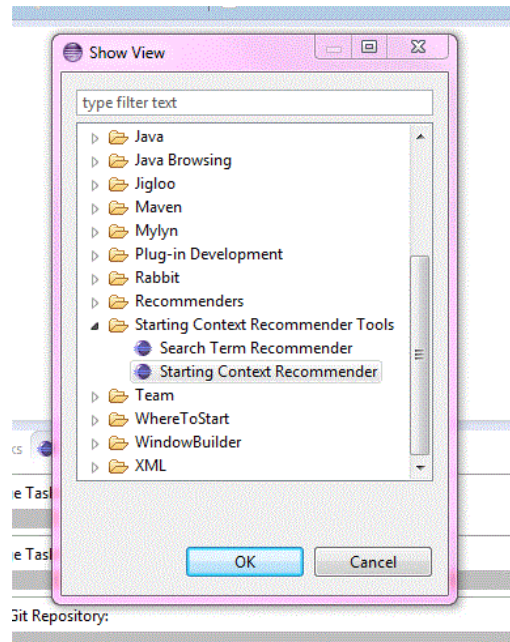


Figure 7.2: The tools included in our Eclipse plugin.

explorer, replace, replace in files, resume, save and save as. Although, we did make use of all of these commands during the explorative study. The keyboard is captured through intercepting keyboard inputs and the selections are captured through the determination of where the cursor is. All gathered data is stored together with a timestamp in XML.

Future Work

Part of future work is the determination of the feasibility of our approach and its prototypical implementation in a user study. Furthermore, many features could extend our approach to automatically identify a starting context for a new change task. In the following, we discuss some of these possible features.

8.1 Presenting the Rationale of a Recommendation

We found in the explorative study, that users may be confused, when they explore a search result which is unclear why it is recommended. Consequently, a mechanism should be included which explains the creation of the search result. As example, the terms which are responsible for the recommendation should be marked. In this way, the user may be more confident about the relevancy of a search result.

8.2 Remembering Relevant Search Results

We observed in the explorative study that some developers had to look for a relevant search result twice, because they searched for other terms in meanwhile. In one case, the participant forgot the term he queried and was spending some of his available time in remembering what he searched for. Thus, our prototypical implementation should include some kind of support to remember relevant search results. One suggestion is to drag and drop relevant search results in a so-called relevant search results *pool*. Search results from this pool can be recovered more easily. Furthermore, functionality to explore the search history would also be an option.

8.3 Including Selections of Prior Searches in the Query

Future work, which could extend a code search tool, is to take prior queries for the change task into account for the generation of the recommendation list for the current query. As example, top results from a prior query which were not further explored, should be excluded from subsequent result lists.

8.4 Examining Task Contexts

Further research can be conducted in the way task contexts are examined. Our work used only textual features to compare a task context to another subject. In a next step, structural features from the code could be included. As example, elements which are reachable within a defined number of navigation steps starting from a task context element could be included in the analysis.

Conclusion

Developers spend a substantial amount of time locating and understanding parts of the source code while performing a change task. In an explorative study we investigated what challenges developers face during the initial search and navigation phase before an actual change is made. Furthermore, we investigated whether the extending of search results with a context supports developers during the initial search and navigation phase. We found that the presentation of a context in the search results supports the developers insofar that search results are more accurately assessed and thus relevant search results were chosen more often. We identified one major challenge in the initial search and navigation phase in finding accurate search terms given a change task description.

Since we found that the contextual information within a search results supports the developers in the initial search and navigation phase, we further investigated the generation of search results by using user interaction histories. We analyzed by using user interaction histories the two steps which are typically employed in an information-retrieval-based procedure: finding similar change tasks and recommending starting points for a change task at hand. We observed that in terms of finding similar change tasks to a change task at hand, the use of user interaction histories does not resolve in an improvement of preciseness. In terms of finding starting point recommendations we could generate for 66.67 % of the analyzed change task a relevant result.

We consolidated the findings of the explorative study and the results on the generation of a starting point in an approach to automatically recommend a starting point for a given change task. We implemented the approach as a prototype to evaluate the feasibility. The user study on the feasibility of our approach is part of the future work.

Participant Instructions

Instructions

In the following you will get 6 change tasks of the project Sando, which you will either investigate using Sando and navigation support of Visual Studio or using CoMoGen additionally to the navigation support of Visual Studio.

For each change task you will find an individual Visual Studio 2012 instance running on the remote computer. The corresponding revision of the Sando solution is already opened in each Visual Studio instance.

Given the change task, please use <Comogen/Sando> to identify three places in the source code (at least on method level) you would point a new programmer to, who will be assigned this task to work on.

During the accomplishment of the change tasks please speak out loud at which elements you are looking at and what you are doing. Try to select what you are looking at.

You are finished with each task when you think you found the relevant elements in the code or when 10 minutes are up. After finishing the accomplishment of the change task you will be asked to fill out a questionnaire about your confidence of the identified starting context.

Please note that your navigation steps within Visual Studio are captured and the screen and audio are recorded.

Appendix B

Questionnaire

Starting Context Confidence Questionnaire

Please write down the three places in the source code you found, that you would point a new programmer to, who will be assigned this task:

Please rate how confident you are that the identified places in the source code are relevant for performing the change task (5-highly confident, ..., 1-not confident).

- ☐ 5
- ☐ 4
- ☐ 3
- ☐ 2
- ☐ 1

Why do you think the places in the source you find are a good starting point?

Please rate how confident you are that the search terms you used yield to good search results (5-highly confident, ..., 1-not confident).

- ☐ 5
- ☐ 4
- ☐ 3
- ☐ 2
- ☐ 1

Please rate the difficulty of the task (5-very difficult, ..., 1-very easy).

- ☐ 5
- ☐ 4
- ☐ 3
- ☐ 2
- ☐ 1

Interview quintessences

number of participants	quintessence
2	The sequence diagram helped me to gain confidence
2	I liked the sequence diagrams
6	I did not like the sequence diagrams
2	I would prefer a list or a tree of search results over sequence diagrams
4	The sequence diagrams were hard to read
9	I assessed a result on the basis of the terms included
4	I think that the code snippets in Sando are useful
4	I ignored the code snippets in Sando
6	I was confused with search result which have no interesting terms in it
2	Comogen misses an outline over all results
4	I think the test classes were useful to display in the search results
3	I think test classes in the search result were not useful
6	The test classes should be marked
3	I think it is better to have less results
10	I tried to select words from the change tasks which can only describe this specific concept

Table C.1: The quintessences of the semi-structured interview (translated from German to English) we conducted with each participant after he finished working on all the task.

Text Preprocessing

Preceding the application of any information retrieval model, text preprocessing steps are conducted in order to achieve a better accuracy of the results. We adapt the five preprocessing steps, which are proposed by Baeza-Yates and Ribeiro-Neto [BYRN99] and include an explicit preprocessing of code related terms to the text features of the change tasks. Some of these preprocessing steps lower the number of terms which are indexed and some of the preprocessing steps come up with term categorization structures. Both types of preprocessing steps are believed to improve the accuracy of the applied information retrieval techniques [BYRN99].

The sequence of the preprocessing steps is rather important. As example, the selection of stop words highly depends on the grammatical status of a term and the exclusion of punctuation requires the previous removal of code snippets and stack traces. Figure D.1 depicts the text preprocessing workflow which specifies which step sequence is conducted.

Lexical Analysis of the Text

A manual screening of different change tasks from the project `org.eclipse.mylyn.context` revealed several discussion points. The identified discussion points and their treatment are enlisted in the following:

- *Code snippets and stack traces.* In most of the change tasks appear code snippets or stack traces as part of the text features. These code related terms need to be extracted from the natural language text, because in the code related terms a strongly diverse vocabulary is used than in the natural language sections. The text of the code related terms holds in our case Java specific vocabulary, which means that symbols have a different meaning allocated than symbols in natural language would have. Thus, we apply different preprocessing steps to the code related terms. See below for details about the preprocessing activities of the code related terms. The subsequent discussion points apply only to natural language text sections.
- *Paths.* Some change tasks include in their text features descriptions of paths, such as `plugins/.mylyn/tasks/<repository>/data/<task id>/.` These paths vary in their appearances throughout the change tasks and do not offer significant semantic information. We exclude these path declarations from our further analysis.
- *Punctuation.* We identified 22 distinct symbols (`?, !, @, :, ", (,), ., [,], <, >, /, *, ', +, -, =, ,, ;, ^, $`), which do not hold significant semantic information and are thus removed from the text and not included in our further analysis. Further, we specified three exceptions of the general removal rule. We think that version specifications and keyboard combinations do hold semantic value which we can exploit. Thus, we do not remove branch specifications,

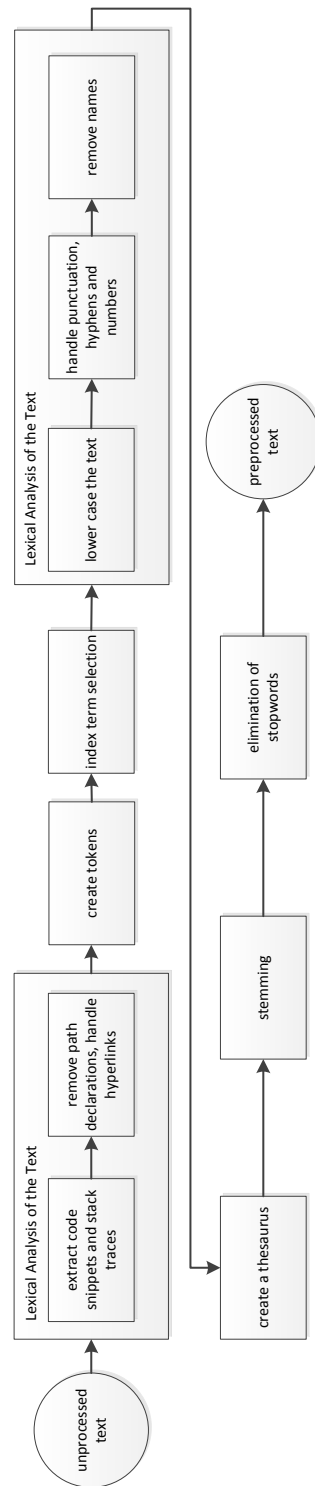


Figure D.1: The sequence in which the text preprocessing steps are accomplished.

such as e_4_3_m_3_9_x, version specifications such as 4.2.2. and keyboard combinations, such as Ctrl+Alt+Shift.

- *Hyphens.* Hyphens are used in different ways throughout the change tasks. On the one hand they are used as part of the term itself, such as example the term *task-specific* and on the other hand they are used as dash, to determine an a time span, as part of an arrow or in an enumeration. We only regard the hyphens which are part of a specific term as valuable, as the splitting of the term would lead often to two terms with little information content.
- *Upper and lower case.* Depending on the position in the text and on the specific type of the term, terms are either lower-cased, upper-cased or a mixture of it. To be able to match terms exactly the entire natural language text is transformed to lower case symbols.
- *Numbers.* In our case, numbers are generally too vague to include strong semantic information and change task ids cannot be distinguished from other six digit numbers. Hence, we exclude numbers from our analysis. As mentioned above, version specifications build the exception from this rule.
- *Hyperlinks.* Many change tasks contain hyperlinks in their text features. These hyperlinks point as example to a Gerrit code review web page ¹, to the Mylyn FAQ web page ², to Hudson logs ³, to the eclipse wiki ⁴, to an eclipse update site ⁵, to tools ⁶, or to the Mylyn download page ⁷. These hyperlinks do not contain significant semantic information. Thus we exclude all of them in our further analysis.
- *Names.* Screening through the change tasks revealed that names of people who contributed to the project or who submitted the change task appear in the text sections. As people's names can be ambiguous and names do not reveal strong semantic information we identified 77 names which are filtered out of the text features.
- *Abbreviations.* Rarely abbreviations of words are used, such as C instead of *see*. Terms which consists only of one letter are removed from the candidate index terms because they are not holding enough semantic information.

Code related Terms Preprocessing

As code related terms can determine a quite exact place in the source code, which is possibly involved in the change task, this information content is important for our analysis. Similar to the text preprocessing steps of the natural language terms, also the code related terms need preprocessing steps to enhance the accuracy of the applied information retrieval approach. The manual screening of the code related terms the following lexical analysis steps:

- *Punctuation.* Code related terms includes a lot of different symbols (:(,), ;, ., =, <, >, ,, \$). In our analysis these symbols are not needed and are thus removed.
- *Numbers.* The included stack traces include a lot of numbers to determine on which line the error occurred. We remove these numbers as they are not holding enough semantic information.

¹as example <https://git.eclipse.org/r/#/c/13879/> in change task 334937

²as example http://wiki.eclipse.org/Mylyn/FAQ#Why_does_startup_of_org.eclipse.mylyn.context.ui_take_so_long.3F in change task 226618

³as example <https://hudson.eclipse.org/hudson/job/mylyn-context-nightly/104/consoleFull> in change task 359547

⁴as example http://wiki.eclipse.org/EGit/User_Guide#Creating_Patches in change task 354989

⁵as example <http://download.eclipse.org/tools/mylyn/update/maintenance> in change task 325551

⁶as example <http://csdl.ics.hawaii.edu/Tools/Jupiter/> in change task 162007

⁷as example <http://eclipse.org/mylyn/downloads/#weekly> in change task 327432

- *Upper and lower case.* The word case is important in Java source code as it indicates which Java construct the term denotes. Thus, we do not preprocess the terms regarding upper and lower case.

Stemming

We stem the terms to their grammatical root, such that we can match terms even when they exhibit plurals, gerund forms or past tenses. This preprocessing step reduces again the number of distinct candidate index terms, which is believed to improve the precision and recall of the applied information retrieval approach. We choose the Porter algorithm to stem the terms of the analyzed change task, as this algorithm applies affix removal which is intuitive and simple [BYRN99].

Elimination of Stop words

Terms which appear in 80% of the change task are regarded as inefficient, because they do not reveal any identifying concepts of the change tasks. As we are analyzing natural language, articles, prepositions and conjunctions appear in every change task and are thus regarded as stop words [BYRN99] which are removed from the candidate index terms. Hence, we filter out 571 distinct english stop ⁸ words from the processed text features. Further, we also add terms to the stoplist, which are specific to the vocabulary used in the analyzed change tasks as Zaman et al. [ZMB11] found out that a tailored list of stop words improves the performance of the applied information retrieval approach.

Removing stop words from the list of candidate index terms, typically compresses the list up to 40% or even more. This reduction of the index terms affects beneficially the precision in the applied information retrieval approaches. In return, the recall degrades if a query includes too many stop words [BYRN99]. As our approach is not dependent on any user input and we use queries which do not include any stop words. So, the risk of reduction of the recall is lowered in our case.

Index Terms Selection

Baeza-Yates and Ribeiro-Neto [BYRN99] argue that most of the significant conceptual information of a text is included in nouns. Hence, one option (option 1) is to select nouns for indexing and not considering other grammatical groups. Broglio et al. [BCCN94] present an approach which clusters nearby nouns and uses this noun group as a distinct index term (option 2). As Baeza-Yates and Ribeiro-Neto suggest, we use a syntactic distance of three as the predefined threshold to build noun clusters [BYRN99]. A third and simple option (option 3) is to select each term which is surrounded by whitespaces. Option 4 to 6 are assembled through adding the preprocessed code related terms to each of the options 1-3.

Thesaurus

An idea can be expressed using several different words. Although the community which contributes to `org.eclipse.mylyn.context` is rather small, it looks like no general glossary is used. Unfortunately, we can only match terms which are composed of exactly the same symbols. To overcome this flaw, we make use of a thesaurus to express a concept with alternative terms. We employed the popular thesaurus of Peter Roget [Com96].

⁸The list of stop words can be found here <http://jmlr.org/papers/volume5/lewis04a/all-smart-stop-list/english.stop>

Contents of the CD-ROM

The following files are stored on the CD-ROM:

- **Zusammenfassung.pdf**
The German version of the abstract of this thesis
- **Abstract.pdf**
The English version of the abstract of this thesis
- **MasterThesis.pdf**
A copy of this thesis
- **StudyData.zip**
All data that was analyzed in connection with the explorative study
- **StartingContextRecommender.zip**
The Eclipse plugin
- **StartingContextRecommender Documentation**
The JavaDoc documentation of the plugin project
- **VisualStudioMonitor.zip**
The Visual Studio extension

Used Libraries, Tools and Plug-ins

- `org.eclipse.mylyn.bugzilla.core`
- `org.eclipse.mylyn.tasks.core`
- `org.eclipse.mylyn.context.core`
- `org.eclipse.mylyn.monitor.core`
- **Jama** - <http://math.nist.gov/javanumerics/jama/>
A linear algebra package for Java
- **Joda-Time** - <http://www.joda.org/joda-time/>
A package to handle date and time in Java
- **Apache Lucene** - <http://lucene.apache.org/core/>
A Java text search engine library
- **The Open Roget's Project** - <http://rogets.site.uottawa.ca/>
Library for lexical analysis resources for natural language
- **Stanford Log-linear Part-of-Speech Tagger** - <http://nlp.stanford.edu/software/tagger.shtml>
Library to determine the part of speech to a word

Bibliography

- [ACCDL00] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance request: a case study. In *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, pages 227–230, 2000.
- [AG05] G. Antoniol and Y.-G. Gueheneuc. Feature identification: a novel approach and a case study. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 357–366, 2005.
- [ASGA12] N. Ali, A. Sabane, Y. Gueheneuc, and G. Antoniol. Improving bug location using binary class relationships. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 174–183, 2012.
- [BCCN94] John Broglio, James P. Callan, W. Bruce Croft, and Daniel W. Nachbar. Document retrieval and routing using the inquiry system. In *In Proceeding of Third Text Retrieval Conference (TREC-3*, pages 29–38, 1994.
- [BMW94] Ted J. Biggerstaff, Bharat G. Mitbender, and Dallas E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, May 1994.
- [BYRN99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Com96] Houghton Mifflin Company. *Roget's II: The New Thesaurus*. Houghton Mifflin Company, 1996.
- [CR01] Kunrong Chen and Vaclav Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 230–, Washington, DC, USA, 2001. IEEE Computer Society.
- [DCR05] R. DeLine, Mary Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 241–248, 2005.
- [EDV05] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 337–346, 2005.
- [EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, March 2003.

- [ES98] K. Erdos and H.M. 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.
- [ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, November 1992.
- [ESW06] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *J. Syst. Softw.*, 79(1):57–68, January 2006.
- [Fie05] Andy Field. *Discovering Statistics Using SPSS*. SAGE Publications, 2005.
- [FKS⁺08] Scott D. Fleming, Eileen Kraemer, R. E. K. Stirewalt, Shaohua Xie, and Laura K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 759–768, New York, NY, USA, 2008. ACM.
- [Fou13] The Eclipse Foundation. Mylyn @ONLINE, August 2013.
- [FTAM96] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliché-based environment to support architectural reverse engineering. In *Software Maintenance 1996, Proceedings, International Conference on*, pages 319–328, 1996.
- [GYK01] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pages 265–274, Washington, DC, USA, 2001. IEEE Computer Society.
- [KAM05] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering, ICSE '05*, pages 126–135, New York, NY, USA, 2005. ACM.
- [KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering, ICSE '07*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
- [KM05a] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development, AOSD '05*, pages 159–168, New York, NY, USA, 2005. ACM.
- [KM05b] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41 – 84, 2005. <ce:title>2003 IEEE Symposium on Human Centric Computing Languages and Environments</ce:title>.
- [KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14*, pages 1–11, New York, NY, USA, 2006. ACM.

- [KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 32(12):971–987, 2006.
- [LK11] Seonah Lee and Sungwon Kang. Clustering and recommending collections of code relevant to tasks. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 536–539, 2011.
- [LPLS86] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
- [MBK91] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *Software Engineering, IEEE Transactions on*, 17(8):800–813, 1991.
- [MJS⁺00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 47–60, New York, NY, USA, 2000. ACM.
- [MKRv05] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubranić. The emergent structure of development tasks. In *Proceedings of the 19th European conference on Object-Oriented Programming, ECOOP'05*, pages 33–48, Berlin, Heidelberg, 2005. Springer-Verlag.
- [MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, July 1996.
- [MRB⁺05] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 33–42, 2005.
- [MSRM04] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, 2004.
- [MTO⁺92] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. *SIGSOFT Softw. Eng. Notes*, 17(5):88–98, November 1992.
- [PPBH91] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research, CASCON '91*, pages 37–53. IBM Press, 1991.
- [RK01] J. Rilling and B. Karanth. A hybrid program slicing framework. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 12–23, 2001.
- [RM02] M.P. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 406–416, 2002.

- [RM03] M.P. Robillard and G.C. Murphy. Automatically inferring concern code from program investigation activities. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 225–234, 2003.
- [RM08] M.P. Robillard and P. Manggala. Reusing program investigation knowledge for code understanding. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 202–211, 2008.
- [RM09] S. Rastkar and G.C. Murphy. On what basis to recommend: Changesets or interactions? In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 155–158, 2009.
- [Saw12] Nicholas Sawadsky. Reverb: Dynamic bookmarks for software developers. Master's thesis, Dept. of Computer Science, University of British Columbia, 2012.
- [SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513 – 523, 1988.
- [SCH98] S.E. Sim, C.L.A. Clarke, and R.C. Holt. Archetypal source code searches: a survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 180–187, 1998.
- [SDVFM05] J. Sillito, K. De Volder, Brian Fisher, and Gail Murphy. Managing software change tasks: an exploratory study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–, 2005.
- [SFH⁺07] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development, AOSD '07*, pages 212–224, New York, NY, USA, 2007. ACM.
- [SLS09] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 157–166, 2009.
- [SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas 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*, pages 21–. IBM Press, 1997.
- [SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, July 2008.
- [TSL03] Christos Tjortjis, Loukas Sinos, and Paul Layzell. Facilitating program comprehension by mining association rules from source code. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension, IWPC '03*, pages 125–, Washington, DC, USA, 2003. IEEE Computer Society.
- [VvMS99] A. Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. Program understanding behavior during corrective maintenance of large-scale software. *Int. J. Hum.-Comput. Stud.*, 51(1):31–70, 1999.
- [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering, ICSE '81*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

- [WGS92] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 200–205, 1992.
- [WGH00] W. Eric Wong, Swapna S. Gokhale, and Joseph R. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54:2000, 2000.
- [WHGT99] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology, ASSET '99*, pages 194–, Washington, DC, USA, 1999. IEEE Computer Society.
- [WS95] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, January 1995.
- [YF02] Yunwen Ye and Gerhard Fischer. Information delivery in support of learning reusable software components on demand. In *Proceedings of the 7th international conference on Intelligent user interfaces, IUI '02*, pages 159–166, New York, NY, USA, 2002. ACM.
- [Zha06] Zhao2006. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, April 2006.
- [ZMB11] A. N K Zaman, P. Matsakis, and C. Brown. Evaluation of stop word lists in text retrieval using latent semantic indexing. In *Digital Information Management (ICDIM), 2011 Sixth International Conference on*, pages 133–136, 2011.