Leveraging Domain Vocabulary across Artefacts: a Comparison of Conceptually Related Applications

Tezcan Dilshener Yijun Yu Michel Wermelinger
Computing Department, Centre for Research in Computing
The Open University
Milton Keynes, United Kingdom

Abstract—**In order to comprehend an unfamiliar application while implementing a change request, software developers refer to all available project artefacts. Although application domain vocabulary plays an essential role in understanding these artefacts, to our knowledge, little has been studied on whether they adhere to the domain vocabulary. In this paper we analyse two applications that implement the concepts defined by the Basel-II Accord between financial institutions. We had access to the user guide, the source code, and some change requests of each application. By analysing the artefacts' vocabulary and the Accord text, we identify the common domain concepts and their correlation to the project vocabulary. We found good conceptual alignment among the artefacts. Searching for relevant classes, given a change request, has shown that change requests or class names do not always explicitly reveal the concepts. Specifically, project conventions cause class names to be overloaded with concept-like terms and may impact the recall and precision of the results, suggesting that leveraging the vocabulary from project artefacts can improve the overall accuracy of the search results and benefit program understanding. We discuss the implication of our results for software maintenance.**

Index Terms—**business software maintenance; domain vocabulary; change requests; empirical study; concept relations.**

I. Introduction

As valuable and strategic assets to companies and playing a central role for the business, software applications require continued and substantial effort to be maintained. From the estimated 80% of overall maintenance efforts, 40-60% is dedicated to comprehension [1]. According to Davison *et al.* [2], even experienced programmers face tremendous learning curves to understand the application domain when they move to work on another area within their current projects. It is argued that during maintenance there is a tendency of 60-80% of time being spent on discovering, which counts towards 20% of overall project costs. Corbi *et al.* [3] identified that understanding a program is an important activity for a developer to perform the designated maintenance tasks and claimed that such activities are treated as transparent and most of the time they go unmentioned.

Often when the original developers left the project a long time ago, software companies turn towards third-party developers to take over the maintenance tasks [21]. The impermanent nature of their job and a high turnover among these developers cause the knowledge of the applications to move further away from its source. Each time a new developer is designated to perform a maintenance task, the associated high learning curve results in loss of precious time, incurring additional costs. As the documentation and other relevant project artefacts decay [11], to understand the current state of the application before implementing a change the designated developer has to go through the complex task of understanding the code. It is stated that humans understand a program when they can describe its characteristics like its architecture, operational context and relations to its domain in qualitative terms other than those used by the program syntax [4]. Comparison between computer and human interpretation of concepts confronts us with human oriented terms, which may be informal and descriptive (e.g. "*deposit the amount into the savings account*"), as opposed to compilation unit terms that tend to be more formal and structured (e.g. "*if (accountType == 'savings') then balance += depositAmount*").

During application development, it would have been ideal for program comprehension to use the same words found in the requirements documentation when declaring identifier names. However, the developers often choose the abbreviated form of the words and names found in the text documentation as well as use nouns and verbs in compound format to capture the described actions [22]. Moreover, the layered multi-tier architectural guidelines better known as Parnas' information hiding principle [5], which advocates the separation of concerns, causes the concept implementations to be scattered across the application. This design principle leads to loss of information and creates challenges during maintenance when linking the application source code to the text documentation. In addition, the separation of concerns coupled with the abstract nature of Object Oriented Programming, obscures the implementation and creates additional complexity for developers during comprehension tasks [6].

Generally, program comprehension during software maintenance can cause additional effort for those developers who have little domain knowledge. Therefore, we are interested to see whether domain knowledge such as the vocabulary of domain-specific concepts could help with program comprehension tasks. To this end, we choose two software applications of the same domain to compare how the domain concepts correlate across various project artefacts. Our aim is to identify the opportunities of using the vocabulary in artefacts to reduce the program comprehension overhead. For instance, the clues obtained from the vocabulary of the application domain can be used to trace a change request to the source code. Compared to existing work on traceability between code and technical documentations, our unique contribution is in using the concepts derived from the trade standard and the user guide. More precisely, we attempt to answer the following research questions:

***RQ1**: What is the adherence of two conceptually related applications to the domain concepts specified by Basel-II?*

***RQ2**: How does the degree of frequency among the common concepts correlate across the project artefacts?*

***RQ3**: What is the vocabulary similarity beyond the domain concepts, which may contribute towards code comprehension?*

***RQ4**: How can the vocabulary be leveraged when searching for concepts to find the relevant classes for implementing change requests?*

***RQ5**: What is the impact on the search results of class names or change requests not reflecting the domain concepts?*

In answering these research questions, we compared the project artefacts of two conceptually related applications addressing the same Basel-II Accord[1] in the finance domain. One of these applications is propriety, whose artefacts vocabulary has been studied in our preliminary work [7]. The other application is open-source, which is fortunate, as datasets of finance domain are seldom made available to the public. This enabled us to perform the designed study. Our analysis helps utilise domain vocabulary across the project artefacts of the two conceptually related applications in program comprehension tasks.

The rest of this paper is organized as follows: Section II describes related work in the literature, Section III describes our data collection procedure and analysis steps, Section IV presents our results in answering the five research questions, discusses their implications to program comprehension and the threats to validity. Finally Section V concludes with additional remarks.

II. Related Work

How vocabulary is distributed amongst the program elements of an application as well as recovering traceability links between source code and textual documentation has been recognised as an underestimated area [8]. It is claimed that due to source code reflecting only a fraction of the concepts being implemented explicitly, one cannot solely rely on the source code as the single source of information [9]. On assessing the relevance of program identifiers in legacy systems, Anquetil *et al.* [10] argued that so-called "*standard*" naming conventions for program identifiers are misleading because conventions imply that an agreement between software developers exists. However, Feilkas *et al.* [11] exposed that the developers follow one another to keep a unified standard in development projects. We investigate further in an environment where recognisable names are used, if the change request and the domain concept terms result in higher quality of traceability between the source code and the text documentation.

Abebe *et al.* [12] present a study on the evolution of the source code vocabulary – words extracted from the comments and code identifiers – for two evolving software applications. The study analysed the history of both applications and measured many attributes. Among the most relevant ones to our study are (1) the type of relations between the individual vocabularies and (2) what the most frequent terms refer to. It is concluded that in case of (1) the vocabulary extracted from the source code file names resulted in having the highest commonality with those found in the comments, and in case of (2) simple frequency analysis of the terms indicated that the frequent terms are related to domain concepts. We also analyse the commonality and the frequency of the vocabulary, but we go further in our investigation: we compare the vocabulary of different artefacts beyond code, and check whether the concepts can be used to map change requests to the source code files to be changed.

In that, our work is similar to the efforts of Antoniol *et al.* [13]. Their aim was to see if the source code classes could be traced back to the functional requirements. The terms from the source code were extracted by splitting the identifier names, and the terms from the documentation were extracted by normalising the text using transformation rules. They created a matrix listing the classes to be retrieved by querying the terms extracted from the text document. The method relied on probabilistic and vector space information retrieval and ranked the documents against a query. Applying precision and recall validated their results. Although the authors compare two different retrieval methods (vector space and probabilistic), they conclude that semi-automatically recovering traceability links between code and documentation is achievable despite the fact that the developer has to analyse a number of sources during a maintenance task to get high values of recall. Our work differs in two main ways. First, it is geared towards maintenance, because we attempt to recover traceability between change requests and source code classes. Second, we improve the precision of the search by using project specific vocabulary and stop-word filtering, instead of grammar stop-words (frequently occurring function words without domain meaning, e.g. ‘a’, ‘be’, ‘do’, ‘for’).

Recently, Abebe *et al.* [14] presented an approach that extracted domain concepts and relations from program identifiers using natural language parsing techniques. To evaluate effectiveness for concept location, they conducted a case study with two consecutive concept location tasks. In the first task, only the key terms identified from the existing bug reports were used to formulate the search query. In the second, the most relevant concept and its immediate neighbour from a concept relationship diagram were used. The results indicate improved precision when the search queries include the concepts. In our study, we also perform concept location tasks using the identified concepts referred by the bug reports, referred to the change requests (CRs). However we have found situations where the relevant concept terms are not adequate enough to retrieve the classes affected by a CR thus we

1. http://www.bis.org

demonstrate the use of CR vocabulary together with the relevant concept terms, additively rather than independently from each other to achieve improved search results.

III. Data Preparation

We analysed the source code of two industrial financial applications, one open source (OSS), referred to as Pillar-One, and one proprietary, due to confidentiality referred to as Pillar-Two. Both applications implement the financial regulations for credit and risk management defined by the Basel-II Accord. Basel-II (otherwise know as Solvency-II) is the second agreement of the Basel Committee on Banking Supervision concerning the international banking laws and regulations. The recommendations of Basel-II are grouped under 3 categories [15]. The first one deals with the management of main financial risk areas such as credit, operational and market risk. The second category focuses on additional risk types like legal and liquidity risk. Finally, the third category provides a framework for dealing with minimum capital requirements in credit business to manage and reduce the financial risk by controlling any risk exposures.

Pillar-One[2] is a client/server application with two main modules developed in Java and Groovy by Munich Re (a re-insurance company). Since Pillar-One's launch in 2008, two additional insurance companies and an open community of developers are involved in the maintenance and further enhancements. The application consists of 5,593 project artefacts comprising source code, configuration, JSP and HTML files and text documentation, which are available from a GIT[3] online repository. The issues, like change requests and problem descriptions, are maintained with JIRA[4], a public tracking tool.

Pillar-Two is a web-based application developed using the Java programming language at our industrial partner's site and is not publicly available. It consists of four modules that are clones of each other: they implement the same business concepts, but differ in how the calculations are performed and parameterized. It has been in production for 4 years and consists of about 2,043 artefacts including source code, configuration files, and user guide documentation. The maintenance and further improvements are undertaken by five developers, including in the past the first author, none of them part of the initial team.

Change requests and preventative maintenance tasks i.e. functionality enhancements and problem corrections for both applications are documented in their respective issue tracking and source control systems. The designated developer is responsible for obtaining the assigned task and searching for the relevant artefacts. So, prior to starting with maintenance, a developer who is new to the technical architecture and vocabulary of these two applications is faced with the challenge of searching the application artefacts to identify the relevant sections. For our source code analysis process, we obtained the complete code, the user guide, and some CRs (27 for Pillar-One and 12 for Pillar-Two). The CRs for both applications are those implemented for the latest production release and are very terse, as shown in Table I.

TABLE I. Sub-set of Change Request Descriptions

CR	Description for Pillar-One
1619	unrecoverable error for error parameter poisson.
1733	Wiring concept for two-phase components should be extended
2024	Handling IllegalAcceessException in Packets and Collectors.
2081	Check usage of enum classes
2093	Quota event limit not implemented.
2163	GIRAModel not compatible with current master branch
2200	NPE when changing result views
CR	**Description for Pillar-Two**
2002	Roundup export data to an importable excel format.
2003	Pdlgd export data to an importable excel format.
2010	Allow volatility values greather than 1.
2063	New reallocation method "Use Asset Diversified Risk".
2068	Show in both sub systems Market and PD/LGD all calculation states
2074	Dialog to distribute lambda factors similar to other module.
2081	Show approx. group values and diversification effects.

Subsequently, we utilised the list of financial domain concepts identified in our previous study [7]. The domain concepts are made up of multiple words (*n*-grams), e.g. “Investment Market Risk”, “Market Value Calculation” and “Lambda Factors”. We took the unique single words (like ‘market’ and ‘lambda’) occurring in the domain concepts as basis for our source code analysis, which resulted in 45 unique concept words. In addition, we have validated the final list of concepts by searching for their occurrences in the official Basel-II documentation.

We processed the obtained project artefacts as described above, using our source code analysis framework tool called ConCodeSe (Contextual Code Search Engine). ConCodeSe is written in Java by extending our previous work and implementing the state of the art data extraction, persistence and search APIs (Lucene[5], Hibernate[6], JIM [16]) to process Java and Groovy source code files. Figure 1 illustrates the extraction, storage, search and analysis stages. In the top layer, the contextual model creation and search services execute the tasks automatically. The left hand side (1) represents the extraction of component words from the source code using the JIM tool and from the textual documentation using the Lucene API. The middle part (2) shows the storing of the extracted vocabulary in the Derby[7] database using the Hibernate persistence API. Finally in the analysis stage (3), the source code analysis over the generated contextual database takes place and the search results are saved in a spreadsheet file for additional source code analysis tasks like the Spearman correlation coefficients tests [17].

ConCodeSe Process Phases

Upon defining the location of the project artefacts and the output in its configuration file, the process (i.e. data extraction, storage and analysis) runs automatically. The ConCodeSe process has two phases. The first phase creates a contextual

2. http://www.pillarone.org
3. https://github.com/pillarone
4. https://issuetracking.intuitive-collaboration.com
5. http://lucene.apache.org/java/docs/index.html
6. http://www.hibernate.org
7. http://db.apache.org/derby

model (corpus) by processing the project artefacts (i.e. concepts, change requests, user guides and source code files) and extracting the component words. In the second phase, the analysis tasks such as searching for the frequency of concepts occurring in the project artefacts and searching for the source files implementing the concepts are performed over the contextual model. The results of the analysis tasks are written to an Excel sheet per task performed.

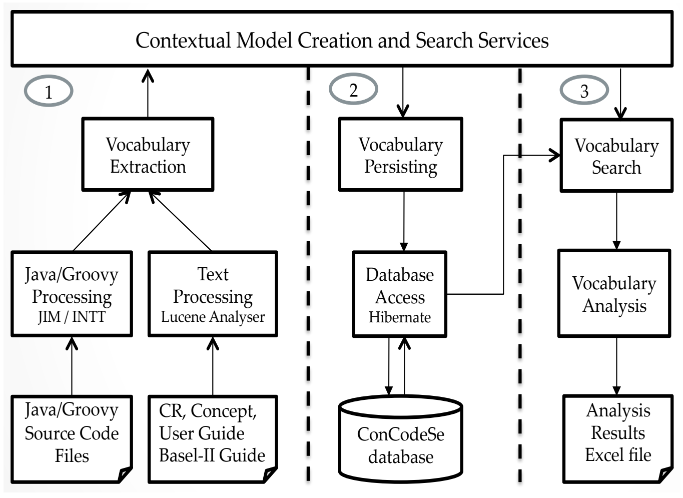
Contextual Model Creation and Search Services
1
2
3
Vocabulary Extraction
Vocabulary Persisting
Vocabulary Search
Java/Groovy Processing JIM / INTT
Text Processing Lucene Analyser
Database Access Hibernate
Vocabulary Analysis
Java/Groovy Source Code Files
CR, Concept, User Guide Basel-II Guide
ConCodeSe database
Analysis Results Excel file

Fig. 1. ConCodeSe Data Extraction, Storage and Search.

Phase I – Contextual Model Creation

In this phase, the project artefact's vocabulary is extracted and saved in two stages: (1) source code vocabulary and (2) textual documentation vocabulary processing.

1. Source code vocabulary processing stage.

In this stage, the first task processes Java and Groovy source code files. The Java source files are parsed using the source code mining tool JIM, which automates the extraction and analysis of identifiers from Java source files. It parses the source, extracts the identifiers and splits them into single terms referred as component words. During this step, the identifiers and metadata from the source code abstract syntax tree (AST) are extracted and added to a central store, with information about their location. Also, the tool INTT [18] within JIM is used to tokenise and split the identifier names into single words. INTT uses camel case, separators and other heuristics to split ambiguous boundaries, digits and lower cases. The extracted information i.e. the identifier names, their tokenisation and source code location, is stored in a Derby database.

In addition, this stage extracts the identifier vocabulary from the Groovy source code files using steps similar to JIM and INTT as described above. The extracted Groovy identifier names and their component words are also stored into the existing database tables whilst retaining referential integrity to existing data. For example, if a component word extracted from a Groovy source file already exists due to its occurrence in a Java source code file then only an additional reference link to its Groovy source code file is created. Otherwise the new word is added to the database with a reference to its location in the source code. The Table II shows the source code artefact size of both applications.

TABLE II. SOURCE CODE ARTEFACT SIZE

	Pillar-One	**Pillar-Two**
Source Code size (LOC)	254,913	80,571
- Class files	2,631	337
- Identifiers	12,257	4,246
- Component words	40,156	13,922
- Unique component words	2,215	683

2. Textual documentation vocabulary processing stage.

This stage extracts the words from the text files i.e. domain concept lists, user guides and change requests. The extracted words are stored in the contextual model. Once again, this process also considers the referential integrity of the existing data, for example if a word extracted from one of the text files already exists in the database due to its occurrence in a Java or a Groovy source code file then a reference link to the word's text file is created. Otherwise the new word is added to the database with a reference to its text file. For this stage, we developed a Java module in ConCodeSe using the Lucene framework to analyse and tokenise the sentences into single words. The module implements the Lucene's Standard-Analyzer class because it tokenises alphanumerics, acronyms, company names, and email addresses, etc. using a JFlex-based lexical grammar. It also includes stop-word removal. We used a publicly available stop-words list[8] to filter them out.

We saved the user guides as a text file to ignore images, graphics, and tables. Confidential information, such as names, email addresses and phone-numbers was then manually removed from the text. Running our Java module over the user guide, we obtained unique words and word instances as shown in Table III. We did the same to extract the words from the change requests, which resulted in a higher number of terms because the CRs are forms containing fields for tracking purposes e.g. 'Priority', 'Assigned Date' and 'Defect Id' are repeated in all change requests. In case of the identified domain concepts, the extraction and storage process resulted in 45 unique concept words with 112 occurrences within the concept descriptions document. Also processing the official Basel-II documentation resulted in 4,726 unique words with a total of 75,292 instances.

In addition, since the CRs were already implemented for the versions of the applications being analysed, we have identified the affected source codes files by manually analyzing the source code repository commits. We have listed the affected source code file names (208 for Pillar-One and 61 for Pillar-Two) as referential link in the contextual model between the CRs and the source code file names. Once all the textual artefacts are processed and component words are stored and cross-referenced, we performed word stemming. For this task, we used Lucene's PorterStemmer class to compute the word's stem by removing the common and morphological endings

8. http://armandbrahaj.blog.al/2009/04/14/list-of-english-stop-words

(a.k.a. inflections). This takes lexical variations into account, e.g. words *calculation, calculated, calculating* etc. have the stem '*calcul'*. We also linked the component words to their stem words in the model. At the end of this phase the contextual model is created with the component words acting as referential link among the artefacts to provide strong relational clues during source code analysis activities as described next.

TABLE III. NUMBER OF WORDS IN TEXTUAL ARTEFACTS

	Pillar-One		Pillar-Two	
Words	**Total**	**Unique**	**Total**	**Unique**
Change Requests	1030	409	1,582	167
User Guide - UG	21,381	3,688	13,801	697

Phase II –Source code analysis.

In the second phase of our process, we developed a search module in ConCodeSe using Java to run SQL queries to (1) search for occurrences of the concepts in all four kinds of artefacts (CRs, user guide, code and Basel-II documentation) and (2) search for all classes that included the component words matching the concepts found in CRs. For each search, we performed exact match and then stem match to see if we obtained more accurate results. In addition, for search (2), the manually identified classes affected by each CR are used to compute precision and recall of the search results. ConCodeSe generates an Excel file with individual work sheets containing information on (1) the contextual model data size (i.e. number of packages, source files, identifiers, number of words etc.), (2) the occurrences of concept words in the project artefacts, (3) the precision and recall of finding the changed classes given the change requests and their concepts, and (4) the word occurrences for each of the project artifact. In ConCodeSe, the results of the search module are measured by *recall:* measures the completeness of the results and p*recision:* measures the accuracy of the results.

The complete processing (i.e. parsing, extracting, splitting, storing, searching and creating the result file) of all four project artefacts (i.e. source code files, change requests and user guides) for both applications took 69 seconds on a dual core iMac with 4GB memory. The resulting database size in case of Pillar-One is 39MB, containing 4,826 unique component words and 3,365 word stems. In case of Pillar-Two, the resulting database size is 14.9Mb, containing 1,195 unique component words and 834 word stems. Although ConCodeSe can process and store multiple artefacts of multiple projects in one contextual model, for our source code analysis purposes we have chosen to keep the models separate for each project.

IV. ASSESSMENT OF RESEARCH QUESTIONS

We searched for the concept occurrences and the vocabulary coverage of the terms across the project artefacts. Using statistical Spearman tests, we demonstrate how the concepts correlate in the two applications, which are independent of one another but yet aim to implement the same regulatory Basel-II Accord. In addition, we show the results of searching for class names matching the concepts referred in the CRs and how a developer leverages vocabulary while executing a maintenance task.

RQ1*: What is the adherence of two conceptually related applications to the domain concepts specified by Basel-II?*

Searching for exact occurrences of concepts in all four project artefacts i.e. CRs, user guide (UG), source code (SRC) and Basel-II documentation (BD – shared by both applications), revealed that while each concept occurred in at least one artefact, only 7 concepts occurred across all artefacts in case of Pillar-One (see Table IV) and only 14 in case of Pillar-Two. Due to space limitations Table IV shows the top seven common concepts, sorted by their frequencies in the CRs, and their respective frequencies in the other artefacts.

TABLE IV. COMMON CONCEPTS IN ARTEFACTS

Concept	**CR**		**UG**		**SRC**		**BD**	
.Pillar-One	**Freq**	**Rank**	**Freq**	**Rank**	**Freq**	**Rank**	**Freq**	**Rank**
time	10	1	35	2	142	3	190	3
current	6	2	15	5	96	4	129	4
capital	3	3	20	3	46	5	1097	2
aggregated	2	4	7	6	42	6	11	7
correlation	1	5	4	7	3	7	38	6
index	1	6	20	4	470	2	42	5
risk	1	7	143	1	1849	1	2196	1
.Pillar-Two								
market	32	1	605	1	561	2	513	2
value	24	2	198	7	481	3	271	3
calculation	14	3	513	2	56	12	102	8
risk	12	4	259	4	411	4	2196	1
asset	8	5	49	11	173	7	190	4
diversified	7	6	37	13	14	13	15	14
base	3	7	104	8	170	8	29	13

Next, we searched again for concepts in the terms and words extracted from the artefacts, but using stemming. This resulted in 4 additional concepts: *label, line, receiver* and *value,* to be retrieved from the Pillar-One project artefacts. In case of Pillar-Two, the stemming did not increase the number of common concepts between the artefacts but it changed the number of instances found. For example, there are 56 exact occurrences of the concept 'calculation' in the source code, but searching for the stem 'calcul' returned 29 additional instances for the terms *calculate, calculated, calculating, calculations* and *calculator*. Also the stemming consolidated concepts *rule* and *rules* into *rule*, which resulted in 44 concepts to be considered during the search.

Furthermore, we noticed that 7 out of 45 concepts resulted in null occurrence in the BD. Upon investigating the reasons, we found that this is due to the naming conventions used in the documentation, for example the concept 'pdlgd' is referenced as 'pd/lgd' and 'subgroup' as 'sub-group' in the BD. Overall, the distribution of the 45 concepts among the artefacts is as given in Figure 2.

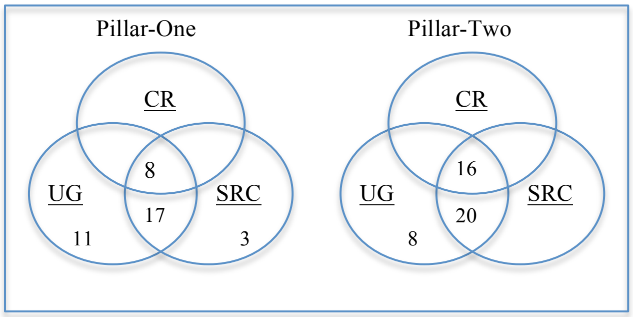

Fig. 2. Concept Distribution among artefacts.

Subsequently, we measured the vocabulary coverage of Basel-II Accord in both applications and searched the occurrence of the Basel-II documentation (BD) words in the three project artefacts i.e. CRs, user guide (UG) and source code (SRC) of both applications. We extracted 4,726 unique words from the BD and 2,771 stems. Once again each word occurred at least in one artefact but only 132 words occurred in all three artefacts in case of Pillar-One and only 49 words occurred in case of Pillar-Two. After stemming, the BD word occurrence in Pillar-One is increased by 16% to 153 words and in Pillar-Two it increased by 35% to 66 words. Since the common vocabulary coverage in all three artefacts is low, 11% and 8% respectively, we wanted to see whether the UG and SRC revealed the terminology used in the guideline drawn out by the Accord. We searched the occurrence of the BD stemmed words in the user guide and source code of both applications. We found identical number of word occurrences in the user guides and source code of both applications. As Table V show, the user guides share 308 common BD words and the source codes share 221 common BD words.

TABLE V. COMMON BD VOCABULARY IN ARTEFACTS

	Exact	Stemmed	UG	SRC
Pillar-One	132	153	308	221
Pillar-Two	49	66	303	221

We expected that the domain concepts identified in our previous study [7] to occur in the official Basel-II documentation (BD). Our search found 38/45 or 87% of the concepts in BD, which supported the validity of the manually identified concepts. Second we anticipated these concepts to exist in the OSS application Pillar-One since it addresses the same domain as well. We searched the occurrences of the concepts and indeed found that the project artefacts of Pillar-One explicitly include the concepts (see Fig. 2), which confirmed that both applications implement the concepts of the highly regulated Basel-II accord. Also, we discovered 4 common concepts: *risk, index, value* and *time* that occurred in all project artefacts of both applications. This indicates that the two independent applications with core functionality for calculating financial risk based on index values over predefined time periods reflect the key domain concepts described in the official documentation.

9. http://www.wessa.net/rankcorr.wasp

RQ2: *How does the degree of frequency among the common concepts correlate across the project artefacts?*

We wanted to identify if the concepts also have the same degree of importance across artefacts, based on their occurrences. For example, among those concepts occurring both in the code and in the guide, if a concept is the n-th most frequent one in the code, is it also the n-th most frequent one in the guide? We applied Spearman's rank correlation coefficient (CC), to determine how well the relationship between two variables in terms of the ranking within each artefactual domain can be described [17]. The correlation was computed pair-wise between artefacts (see the Table VI headings), over the instances of the concepts common to both artefacts. Table VI shows the results obtained using the online Wessa statistical tool[9].

In case of Pillar-One, for both search types (exact and stem), the correlation between CRs and the other three artefacts (UG, SRC and BD) is low and not statistically significant (p-value > 0.05). This is because there are very few common concepts with exact occurrences in CRs. However, the correlation among UG, SRC and BD without CRs is stronger and statistically significant (p-value < 0.05). For stem search, the correlation decreases in most cases.

TABLE VI. SPEARMAN CORRELATION OF CONCEPTS AMONG ARTEFACTS

Correlation Between =>	**CR vs UG**	**CR vs SRC**	**CR vs BD**	**UG vs SRC**	**UG vs BD**	**SRC vs BD**
P1- Normal						
Correlation	0.169	-0.036	0.214	0.884	0.848	0.643
Probability	0.674	0.928	0.596	0.030	0.037	0.114
P1- Stem						
Correlation	-0.175	-0.018	0.073	0.782	0.536	0.207
Probability	0.575	0.952	0.818	0.013	0.089	0.509
P2- Normal						
Correlation	0.414	0.183	0.558	0.525	0.418	0.421
Probability	0.133	0.503	0.043	0.057	0.131	0.128
P2- Stem						
Correlation	0.635	0.178	0.664	0.553	0.574	0.298
Probability	0.021	0.516	0.016	0.046	0.038	0.280

In case of Pillar-Two for exact search, the correlation is low and not statistically significant except between CR and BD as well as UG and SRC. For stem search, the correlation between CR and UG as well as UG and BD becomes stronger and statistically significant. On the contrary to Pillar-One where stemming increased the number of matched words, the stemming of words into their root increases their instances in the artefacts. This changed the relative ranking of the common concepts. The Spearman correlation became stronger and statistically more relevant, as Table VI shows. Only the correlation between CRs and SRC as well as SRC and BD remains insignificant.

We found that only 25/45 or 56% of concepts occur both in the code and documentation in case of Pillar-One and 36/45 or 80% in case of Pillar-Two. Since the source code is consulted during maintenance, this lack of full agreement between both artefacts may point to potential inefficiencies during maintenance. However, using stemming to account for lexical variations, the common concepts correlate well (with a high statistical significance of $p=2*10^{-4}$) in terms of relative frequency, taken as proxy for importance, i.e. the more important concepts in the user guide tend to be the more important ones in the code. This good conceptual alignment between documentation and implementation eases maintenance, especially for new developers. The weak conceptual overlap and correlation between CRs and the other two artefacts is not a major issue for us since CRs are usually specific for a particular unit of work and may not necessarily reflect all implemented or documented concepts.

RQ3: *What is the vocabulary similarity beyond the domain concepts, which may contribute towards code comprehension?*

We searched the contextual model of Pillar-One using the source code words of Pillar-Two, excluding the concept words. We identified 428 common words in both applications with varying frequency. We also cross-checked the validity of the identified common words by repeating the search over the contextual model of Pillar-Two using the source code words of Pillar-One and obtained the same number of common words. The common words search results indicated that 63% of Pillar-Two's vocabulary exists in Pillar-One and conversely only 18% of Pillar-One's vocabulary exists in Pillar-Two.

Subsequently, we wanted to find out if the common vocabulary also contributes towards ease of code understanding by developers. Since identifiers are made up of compound words, we looked at overall vocabulary of all the identifiers and not just component words that occur in the source. After all, even if there is overlap in the component words, one application might combine them in completely different ways to make identifiers that other developers will find confusing. Indeed, both applications combine the words differently as we found only 37 common identifiers between the applications. As shown in Table VII, the concept 'Market Value' is used in Pillar-One on the left-most position of an identifier name, whereas the same concept is used in Pillar-Two on the right-most position. Based on our experience of project naming conventions, it is found that method parameters are prefixed with "p" as in "pBase" compared to "parmBase" in Pillar-One.

TABLE VII. SAMPLE OF IDENTIFIER SIMILARITY

Pillar-One	marketValueOfBods	getLabelText	parmBase
Pillar-Two	longTermMarketValue	getLabelName	pBase

Since both applications are about the same domain, we wanted to find out if the developers have some common vocabulary that is outside the domain concepts. We analysed the vocabulary coverage of the BD among the project artefacts of both applications and found that the common vocabulary coverage reveals the developers' compliance with the official documentation (see Table V). Although both applications combine the words differently on the identifier names (only 37 common ones), comparing the rest of the identifiers based on our industrial experience with Pillar-Two, revealed similarities indicating that the developer of one application may easily navigate the code base of the other one (see Table VI). This indicates two good pointers for maintenance (a) full business vocabulary coverage, and (b) in both projects abbreviations are not required to retrieve such vocabulary from the artefacts.

RQ4: *How can the vocabulary be leveraged when searching for concepts to find the relevant classes?*

Next, we searched each application for class names matching the concept words referred by the CRs. We started our experiments with Pillar-Two first because we were able to identify the concepts addressed by each CR based on our industrial experience hence allowing us to accurately validate and calibrate the search techniques within our tool.

RQ4.1. Pillar-Two:

For each search, we performed exact match and then stem match to see if we obtained more accurate results. We computed the precision and recall of the results by comparing the retrieved classes to those that should have been returned, i.e. those that were affected by implementing the CR. Exact CR concept search had very high recall with very low precision. Since a stemmed search returns a superset of exact search, we expected it to deteriorate precision and to improve recall. Table VIII shows the subset results of the stemmed search; indeed, the precision did deteriorate, e.g. for CR #2074 it declined from 19.05% to 16.67%. The reason for this is that the stem of 'factors' is 'factor' and of 'value' is 'valu', resulting in 8 additional classes to be retrieved. The stemmed search did not improve recall either, e.g. no additional relevant classes were found for CRs #2002 and #2003 thus precision and recall remained 0%.

TABLE VIII. PILLAR-TWO STEM CONCEPT SEARCH RESULTS (SUBSET)

CR #	stemmed concepts searched	relevant classes	relevant retrieved	recall (%)	retrieved classes	precision (%)
2002	roundup, valu	15	0	0	11	0.00
2003	valu	6	0	0	11	0.00
2010	volatil, market, valu	6	6	100.00	142	4.23
2063	index, asset, market	7	6	85.71	161	3.73
2074	factor, lambda, valu	6	4	66.67	29	16.67
2081	group,roundup,helpr	6	0	0	11	0

We looked further at the reasons for low precision. In the case of CR #2002 (0% recall and precision), the request is about a generic action (exporting) (see Table I) on the concept (roundup), and as such the concept does not appear in the relevant class names. Other CRs involve the frequent concept 'market' (see Table IV), which due to the project naming conventions occurs in almost every class name of the module, causing many false positives.

We asked an additional research question: *How can we tailor the search to improve precision, so that developers have to inspect fewer classes for relevance?*

We prepared a customized search for a subset of the CRs from Table I. First, we searched the classes using the actual words of the CR, rather than its associated concepts, because they better describe the concept's aspects or actions to be changed. However, the CR and the class identifiers may use different words. For example, the CR #2081 term 'mask' refers to the GUI, which is implemented by the Helper pattern, explicitly referred to in class names. Hence we introduced a project specific mapping mechanism, which in our case includes 'mask' $\rightarrow$ 'helper'. Finally, we discarded from searches project specific stop-words by automatically selecting the two most frequently occurring concepts, 'market' and 'value' for Pillar-Two and 'time' and 'current' for Pillar-One (Table IV). The new results obtained are shown in Table IX. We see that precision increased by 100% for CRs #2010 and #2063 compared to Table VIII, while the impact on recall remained minimal and the previously not detected classes for CR #2002, #2003 and #2081 are also retrieved.

TABLE IX. PILLAR-TWO SEARCH RESULTS WITH STOP WORDS (SUBSET)

CR #	CR vocabulary searched	relevant classes	relevant retrieved	recall (%)	retrieved classes	precision (%)
2002	roundup,data,export	15	15	100.00	74	20.27
2003	pdlgd,data,export,…	6	6	100.00	79	7.59
2010	volatility,…	6	5	83.33	81	6.17
2063	asset,diversified,…	7	5	71.43	81	6.17
2074	factor, lambda, …	6	4	66.67	95	4.21
2081	group,roundup,helpr	6	1	16.67	35	2.86

RQ4.2. Pillar-One:

In case of Pillar-One, we repeated the same search tasks as for Pillar-Two (exact match followed by stem match) and obtained 0.00% recall and precision. This indicated that either the action-oriented nature of CRs did not indicate clues to the concepts being addressed directly or the class names of Pillar-One did not reflect concept words explicitly. So we searched again using the CR vocabulary and also, we assigned a weight to each resulting class based on the query words matching the terms found in a class name. For example, given a query with words "*abc*" and "*xyz*", if the found classes are *AbcXyzClass* and *XyzClass*, then the class weights are 2 and 1 respectively because class *AbcXyzClass* has both of the queried words in its name. The weight is used to rank the classes in the result list, which allowed us to focus only on the top 30 classes since the developers are unlikely to go through any longer lists. We found relevant classes matching 5 CRs as Table X shows.

TABLE X. PILLAR-ONE SEARCH RESULTS WITH CR WORDS (SUBSET)

CR #	CR vocabulary & concept searched	relevant classes	relevant retrieved	recall (%)	retrieved classes	precision (%)
1619	lambda	24	0	0	0	0
2093	quota,feature,event	6	2	33.33	4	50.00
2163	master,comp,plugin	60	6	10.00	15	40.00
2169	section,config,pane	14	6	42.86	13	46.15
2200	crash,open,aggregat	13	6	46.15	24	25.00
2081	quota,speific,feature	6	2	33.00	4	50.00

The improvement in recall inspired us to experiment further by repeating the search but this time considering all of the words extracted from the identifiers contained in each class in addition to the words extracted from the class name. For this task we created an indexed contextual model of class names and all their associated words using Lucene's Vector Space Model (VSM). In VSM, each document and each query is mapped onto a vector in a multi-dimensional space, which is used for indexing and relevance ranking [19]. In our case, each class with its list of words is treated as a document in the VSM. We searched the indexed "class by words" document model using Lucene's query API to take full advantage of the VSM. As shown in Table XI, exact search discovered the classes for CR #1619. The stem search did not expose any additional classes, except for a slight drop in precision values due to the stem returning a superset of exact search as described earlier. The search with CR vocabulary in addition to concept words resulted in discovering classes for 2 additional CRs #1733, #2024. Overall, we recovered traceability links between the CRs and the affected classes for 8 CRs by combining the vocabulary of the project artefacts.

TABLE XI. PILLAR-ONE SEARCH RESULTS USING VSM (SUBSET)

CR #	Concept searched Vocabulary srchd	relevant classes	relevant retrieved	recall (%)	retrieved classes	precision (%)
1619	lambda	24	2	8.33	13	15.38
1733	multi,phase,comp…	8	1	12.50	276	0.36
2024	save,collect,value,...	4	3	75.00	163	1.84

We searched for the classes using the concepts referred by the CRs and obtained differentiating results. We found that in case of Pillar-Two, mapping a CR's wording to domain concepts and using those to search for classes to be changed, is enough to achieve very good recall, but precision is poor and in case of Pillar-One both recall and precision remains low. Upon investigating the reasons for this, we discovered that in Pillar-Two the class names reflect the concepts and the CR descriptions have very good concept word coverage while this is not the case in Pillar-One. Also we found that overloading the class names with concept like project specific words cause high recall and low precision on one hand and on the other hand having no indication to the concepts referred by a CR produce no results. Since CRs may come from a group of people who are unfamiliar with the vocabulary used in the code and in the documentation, we recommend CR documents to contain a section for describing the relevant concepts that the current CR is referring to.

RQ5: *What is the impact on the search results of class names or CRs not reflecting the domain concepts?*

Since we were able to recover at least one traceability link for all of Pillar-Two CRs but not for Pillar-One, we have investigated the reasons for this and found out that Pillar-Two class names and CR descriptions reflect the concepts more explicitly than Pillar-One as illustrated in the following example. Table XII shows two CRs concerning the concept "*lambda*". In case of Pillar-Two CR #2074, the four classes are found because these class names include the searched concept.

However, in case of Pillar-One CR #1619, the affected classes are not detected during the exact concept search because the class names do not include the concept word being searched. After extending the search, in case of Pillar-One, to also consider the identifier names existing within the body of a class, the exact search found the affected classes based on the identifier names.

TABLE XII. CLASS NAMES AND IDENTIFIERS EXAMPLE

CR	Concept	Affected Classes	Identifier
2074 (P2)	factors, lambda, value	K4MarketHelperDistributeLambda K4MarketRiskLambda, K4MarketProcessorCopyLambda, K4MarketHelperCopyLambda	calculateLambdas, readLambda, hasLambdaDiversified, lambdaFactorSet
1619 (P1)	lambda	LognormalTypeIIParetoDistributio TypeIIParetoDistribution	alphaAndLambda, muAndLambda

Furthermore, in case of Pillar-One, the recall and precision values for exact and stemmed search results were 0.00% (see RQ4.2). Upon investigating the reasons we found in addition to class names not reflecting the concepts, the terse CR descriptions also fail to reflect most of the concepts (see Table I). This is also illustrated in Fig. 2 where only 8/45 or 18% of the concepts occur in Pillar-One CRs, while Pillar-Two's CRs include double that amount.

At this stage we asked another research question: *How would a developer leverage the quantitative information to comprehend an application?*

We asked a Pillar-Two developer to re-implement Pillar-One CR# 2031 "*Net reserve risk uses gross numbers*". To obtain a list of candidate classes as initial starting point, he compared the CR terms against the concept occurrences (see Table IV) and searched for the class names containing the term "risk", which returned no results as the class names in Pillar-One do not contain the concept "risk". So he considered additional CR terms based on their ranking in the source code obtained by searching the occurrences of the Basel-II documentation (BD) vocabulary across the project artefacts of Pillar-One in answering the RQ1 (see Table XIII).

TABLE XIII. BD VOCABULARY OCCURRENCES IN PILLAR-ONE (SUBSET)

BR words .Pillar-One	CR		UG		SRC	
	Freq	Rank	Freq	Rank	Freq	Rank
reserve	11	1	110	13	309	19
gross	2	23	135	3	107	30
numbers	1	56	49	23	48	41

The next candidate CR term to use was *reserve*, which resulted in 57 classes. Next, he decided to narrow down the search to include those classes that are of relevance for the CR. He extended the search to also consider the terms contained in the body of the classes. Based on Table XIII, he selected the next highest ranked SRC term *gross* from the CR description to be searched in the body of the candidate classes, which narrowed the results to only 6 classes. He considered 4 of them to be irrelevant as their names indicated these classes to be configuration related and decided to focus on the remaining two as candidate classes. Investigating the body of the two classes revealed program logic for calculating "gross claims" and instances of the term *gross* occurring together with the number 1 ranking term *claim*. Subsequently, he continued by searching for the classes referring to the two candidate ones but we decided not to investigate further since our aim was only to demonstrate the use of quantitative analysis to aid developers in discovering the relevant classes.

Both recall and precision can be improved by (a) using the actual CR vocabulary, (b) mapping some of it to different terms used in class identifiers, (c) ignoring frequent concepts, which act as stop-words and (d) considering all the words extracted from a class during the search. We note that such project specific, simple, and efficient techniques can drastically reduce the false positives a developer has to go through to find the classes affected by a CR. We also note that in projects like this, where class identifiers are more descriptive than abbreviations, the use of stem search decreases precision, while not increasing recall. In addition we observed that developers would benefit if CRs provide a section with ranked keywords to be used when searching the relevant program elements during maintenance.

Threats to Validity

A single developer demonstrated the use of quantitative analysis results in discovering the relevant classes. This threat to internal validity will be addressed by conducting controlled experiments involving additional developers. We only used single-word concepts, while business concepts are usually compound terms. We searched the project artefacts of Pillar-One using the concepts defined in our previous study [7]. In fact, it is feasible to have other concepts that are not reflected in our list. Also, the relative frequency was taken as the proxy for importance.

These threats to construct validity will be addressed in future work: we will see if term co-occurrence improves precision and will elicit concepts from the official Accord. The characteristics of the projects (the domain, the terse CRs, the naming conventions, the kind of documentation available) are a threat to external validity. We catered for this by repeating the experiments with two different applications addressing the same financial domain for comparison. In our future work we aim to consider additional applications from other domains.

V. CONCLUDING REMARKS

In this study we investigated the vocabulary of two independent financial applications covering the same domain concepts. We found that artefacts explicitly reflect the domain concepts and the paired artefacts of these applications have good conceptual alignment. This alignment helps facilitate program comprehension when searching for classes affected by given change requests (CRs). Both applications adhere with the vocabulary of the Accord and reveal good overlap in-between, indicating a full coverage of the business vocabulary. Although the descriptive identifiers support high recall of classes affected by CRs, the low precision may be detrimental to maintenance. CR descriptions or class names do not always reveal the concepts, and class names are found overloaded with concept terms due to naming conventions, which may account for the high recall/low precision phenomena.

The importance of descriptive and consistent identifiers for program comprehension has been extensively argued for in the academic [8] and professional literature [20]. Our study showed that despite good naming conventions and vocabulary coverage, it is challenging to find the classes referred by a CR. Although reflecting the daily tasks of the developers, CRs of action oriented nature are not a better source for traceability than user guides and trade standards. It is found that these two types of documentations describe the usage scenarios and may account for the improved traceability.

This work is an exploration of the vocabulary relations between artefacts and the concepts. It illustrates that even in highly regulated environments naming conventions cause overloading and overcloud program comprehension, hence providing further evidence that one cannot rely on the code as the single source of information. This in turn highlights the need for better programming guidelines and tool support beyond enforcing naming conventions within the code because naming conventions alone do not guarantee a good traceability between the concepts and other project artefacts. In our future work, we will consider constructing a contextual model as the consistent set of clues to aid program comprehension.

ACKNOWLEDGMENT

We thank our industrial partner in Germany, for providing the artefacts and their input on diverse information required. We also thank Simon Butler for his suggestions on an earlier draft of the paper.

REFERENCES

[1] G. Pigoski, T. M.: “Software maintenance”; In: Guide To The Software Engineering Body Of Knowledge; Los Alamitos, CA: IEEE Computer Society Press. Trial Version 1.00, May, 2001.

[2] J. W. Davison, D. M. Mancl, and W. F. Opdyke, “Understanding and addressing the essential costs of evolving systems,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 44-54, 2000.

[3] T. A. Corbi, “Program understanding: Challenge for the 1990s,” *IBM Systems Journal*, vol. 28, no. 2, pp. 294 -306, 1989.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” in *Proceedings of the 15th Int’l Conf. on Software Engineering*, 1993, pp. 482-498.

[5] D. Parnas, "On Criteria To Be Used in Decomposing Systems Into Modules," *Communications of the ACM*, 14(1):221-227, April 1972.

[6] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards Supporting On-Demand Virtual Remodularization Using Program Graphs", AOSD 2006, ACM, pp. 3-14.

[7] T. Dilshener and M. Wermelinger, “Relating developers’ concepts and artefact vocabulary in a financial software module,” in *Software Maintenance, 2011 27th IEEE Int’l Conf. on*, pp. 412-417.

[8] F. Deissenböck and M. Pizka, “Concise and consistent naming,” in *Proc. 13th Int’l Workshop on Program Comprehension*, 2005, pp. 97– 106.

[9] F. Deissenböck and D. Ratiu, “A unified meta-model for concept-based reverse engineering,” in *In Proceedings of the 3rd Int’l Conf. Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM’06)*, 2006

[10] N. Anquetil and T. Lethbridge, “Assessing the relevance of identifier names in a legacy software system,” in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, 1998, pp. 213-222.

[11] M. Feilkas, D. Ratiu, and E. Jurgens, “The loss of architectural knowledge during system evolution: An industrial case study,” in *Program Comprehension, 2009 IEEE 17th Int’l Conf. on*, pp. 188-197.

[12] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, “*Analyzing the Evolution of the Source Code Vocabulary*,” *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pp. 189–198.

[13] G. Antoniol, G. Canfora, G. Casazza, A.D. Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE Transactions on Software Engineering*, 28:970-983, 2002.

[14] S. L. Abebe and P. Tonella, “Natural Language Parsing of Program Element Names for Concept Extraction,” in *Program Comprehension, 2010 IEEE 18th Int’l Conf. on*, pp. 156-159.

[15] Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version, June 2006, http://www.bis.org/publ/bcbs128.htm.

[16] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, “Exploring the influence of identifier names on code quality: an empirical study,” in *14th European Conf. on Software Maintenance and Reeng.*, 2010, pp. 159–168

[17] S. Boslaugh, P. Watters, “Statistics in a nutshell”, O’Reilly, 2008, pp. 176-179.

[18] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, “Improving the tokenisation of identifier names,” in *Proc. European Conf. on Object-Oriented Programming*, LNCS 6813, Springer-Verlag, 2011, pp. 130-154

[19] G. Salton and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing &; Management*, vol. 24, no. 5, pp. 513-523, 1988.

[20] R. C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”, Prentice Hall, 2008, pp. 17-30.

[21] K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 73-87.

[22] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “What is in a Name? A Study of Identifiers,” *Proc. 14th Int’l Conf. on Program Comprehension*, IEEE, 2006, pp. 3-12