Integrating Defect Data, Code Review Data, and Version Control Data for Defect Analysis and Prediction

by

Tao Chun Lee

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
at the
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (EPFL)
August 2013

Author .....
School of Computer and Communication Sciences
August 29, 2013

Certified by .....
George Candea
Professor
Thesis Supervisor

Certified by .....
Ronny Kolb
Director of Engineering Productivity, Honeywell ACS
Thesis Supervisor

Certified by .....
Jose Parra
Quality Manager, Honeywell ACS
Thesis Supervisor

Integrating Defect Data, Code Review Data, and Version Control Data for Defect Analysis and Prediction

by

Tao Chun Lee

Submitted to the School of Computer and Communication Sciences
on August 29, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In this thesis, we present a new approach to integrating software system defect data: Defect reports, code reviews and code commits. We propose to infer defect types by keywords. We index defect reports into groups by the keywords found in the descriptions of those reports, and study the properties of each group by leveraging code reviews and code commits. Our approach is more scalable than previous studies that consider defects classified by manual inspections, because indexing is automatic and can be applied uniformly to large defect dataset. Also our approach can analyze defects from programming errors, performance issues, high-level design to user interface, a more comprehensive variety than previous studies using static program analysis.

By applying our approach to Honeywell Automation and Control Solutions (ACS) projects, with roughly 700 defects, we found that some defect types could be five times more than other defect types, which gave clues to the dominant root causes of the defects. We found certain defect types clustered in certain source files. We found that 20%-50% of the files usually contained more than 80% of the defects. Finally, we applied a known defect prediction algorithm to predict the hot files of the defects for the defect types of interest. We achieved defect hit rate 50%-90%.

Thesis Supervisor: George Candea
Title: Professor

Thesis Supervisor: Ronny Kolb
Title: Director of Engineering Productivity, Honeywell ACS

Thesis Supervisor: Jose Parra
Title: Quality Manager, Honeywell ACS

Acknowledgments

Je n'avais d'abord projeté qu'un mémoire de quelques pages; mom sujet m'entraînant malgré moi, ce mémoire devint insensiblement une espèce d'ouvrage trop gros, sans doute, pour ce qu'il contient, mais trop petit pour la matière qu'il traite.

J. J. Rousseau, préface, Emile ou de l'Education

My most humble and sincere thanks to:

My professor at EPFL, George Candea, for his remote but powerful guidance. George's unparalleled knowledge in computer systems shaped this project from the very beginning. George's passion inspired me to carry out this industry-based project with academic rigor. George's idea of mixing in machine learning to this project proved to be very fruitful. I learned a lot from George, both in computer science and life. Thank you, George, for this enjoyable ride.

My supervisor at Honeywell ACS, Ronny Kolb, for his constructive advice. Ronny's advice was critical to give an industrial taste to this project. Doing a project with both academic rigor and industrial practicality was difficult, and Ronny provided the input from the industry side. Vielen Dank, Ronny.

My supervisor at Honeywell ACS, Jose Parra, for his weekly supervision. Jose's role in this project was more than important. He was the one overseeing the weekly progress of this project, he listened to my complaints and difficulties to the finest details and he always helped me in his humorous way. Jose's and I shared common interest and excitement in indexing defects by keywords, and he supported this idea from the very beginning, especially in difficult times when this new idea encountered bottlenecks. I thanked Jose for his support and constant feedback. Muito obrigado, Jose.

My EPFL mates and des professeurs de français, Youssef Ait Khalifa and Etienne Helfer, for their daily input. Youssef and Etienne acted as sounding boards of my premature ideas. Youssef taught me the French language in his unique way by introducing me to *France 24*, the online French news media that became my daily French news source ever since. I learned more from him than he ever knew. Etienne showed me his interest in simulation games. We shared common interest in the classical simulation game, *SimCity*, which became our hot topic of discussion ever after. Merci beaucoup, Youssef et Etienne.

My friends on the logistic team and the production team of Honeywell ACS, Amir Sadriu, Jennifer Freeman, Kevin Naegeli, Yanick Moukolo and Simon Cawsey, for their fun sharing during lunch time and train hours before & after work.

My administrative support from Honeywell ACS and EPFL, Véronique Hatfield, Yasmina Hepp, Nicoletta Issac and Antonella Martin-Veltro. Véro helped me settle down and get around Honeywell ACS from the very first day. Véro taught me a lot about the French language, and how to be courageous to speak French without worrying too much. Yasmina helped me with engagement procedure. Nico provided assistance for the administrative issues in Dependable Systems Laboratory. Antonella provided support from EPFL, by reminding all the requirements of doing a PDM in the industry and helping me keep pace.

My hosting company, Honeywell ACS, for giving me this opportunity to experience a world-class enterprise producing diversified products in the area of automation and control solutions. I was more than happy to learn this business committed to helping more people around the world.

My school, EPFL, for teaching the necessary knowledge to solve real problems and for enabling me to pursue a career in cutting-edge technology. I was more than fortunate to be educated at EPFL, one of the best engineering schools in this world.

My family and friends in Taiwan and Switzerland, for their distance-invariant love, support and encouragement over the years.

Cette page est trop petite pour nommer toutes les personnes qui ont contribué à ce travail. Qu'elles sachent ma gratitude et reçoivent mes tout remerciements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> My name is Sherlock Holmes. It is my business to know what other people do not know.
>
> Arthur Conan Doyle, The Adventure of the Blue Carbuncle

This thesis examines the integration of the three-dimensional defect data found in defect reports, code reviews and code commits. We attempt to collect information from the defect dataset and address questions like: What are the defects? What defects are discovered in code reviews? How are defects distributed in files? Do defects cluster in files? How to predict the hot spots of defects? How do different projects compare?

1.1 Motivation

Defects are the unwanted side effects of software development: they deteriorate the performance, compromise the functionality, and threaten the security and safety of the systems. With the use of software engineering tools, large defect dataset is collected and stored, and we want to leverage this dataset for defect analysis and prediction. The dataset under consideration, as illustrated in Figure 1-1, includes 1) defect re-

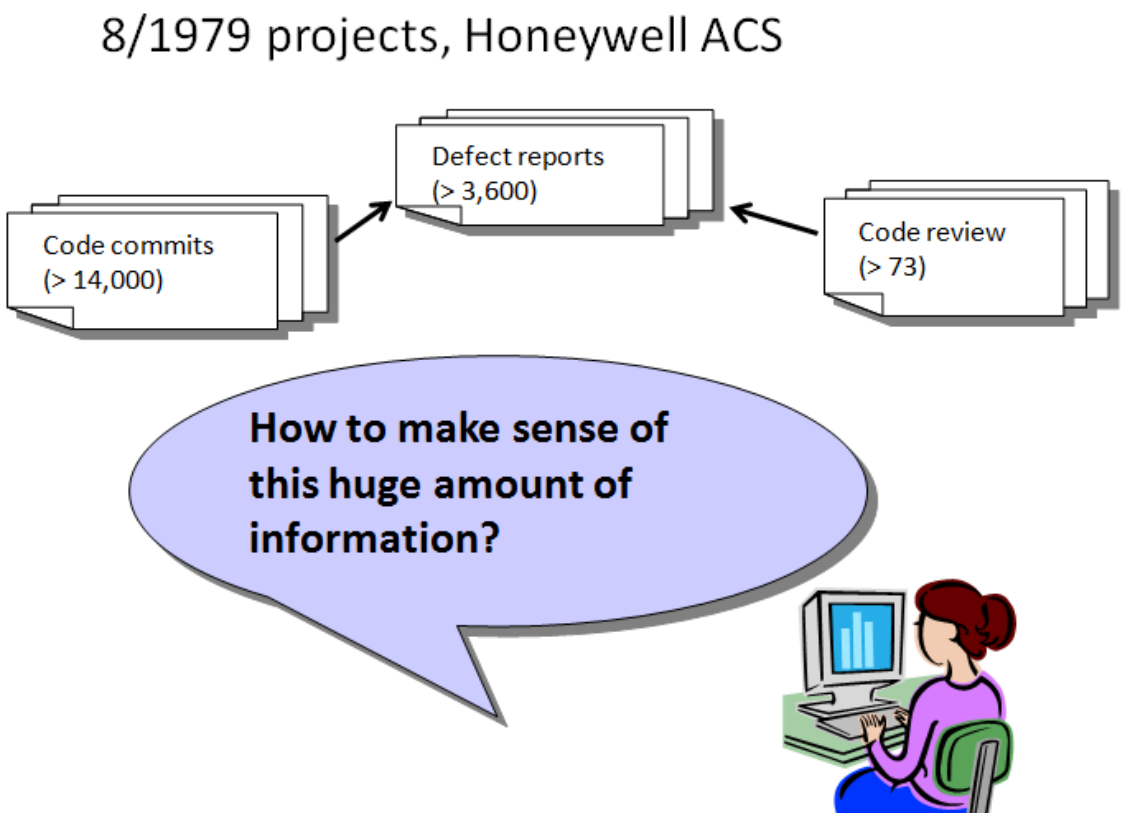


Figure 1-1: The big defect data challenge

ports, the descriptions of the defects, 2) code reviews, the reviews of the code changes by programmers, and 3) code commits, the file changes of each code commit.

Honeywell ACS, like many other enterprises, uses software engineering tools to collect defect data in defect reports, code commits and code reviews. Over the years, huge defect dataset is saved in database, and it is an open question how to leverage this dataset. This thesis is motivated by this big defect data challenge, and attempts to find a solution to solve this challenging problem.

In our experience, the biggest obstacle to defect analysis and prediction is not lacking analysis tools nor inaccuracy of prediction models. Program analysis tools can scan codebases and find many errors and warnings. Prediction models can model the evolution of the identified defects with acceptable accuracy. Instead, the biggest obstacle to analyzing and predicting defects is simply knowing what are the defect types. Program analysis tools can capture mostly non-functional defects, which only

accounts for a small fraction of the discovered defects. Manually discovering any defect types is a discouraging adventure, especially when it must be carried out for many projects.

In many Honeywell projects, a defect report provides an optional field to specify its defect type. Unfortunately, this optional field is very often either left unspecified or specified inappropriately by choosing it from a set of predefined types. The end result is an ad hoc collection of defect types without much meaningful information. Since manually finding defect types is difficult, we instead focus on techniques to automatically extract defect types from defect reports without prior knowledge of the projects.

Basing our results on machine-inferred defect types has two advantages. First, the machine can extract defect information from every defect report, and can analyze defects from programming errors, performance issues, high-level design to user interface, a more comprehensive variety than previous studies using static program analysis. Second, the machine scans the defect dataset uniformly, making the approach more scalable to large dataset compared to manual inspections.

The scope of defects used in this study is limited to defects found in defect reports. These defects, though, cover many facets of a complete system from programming practices, performance issues, high-level design to user interface, they tend to over-represent defects where skilled developers happened to look or defects happened to be triggered most often. It is an open question if our conclusions will hold for defects not found in defect reports.

1.2 Research questions

While integrating the three-dimensional dataset of software system defects is a broad topic, we focus on several key research questions emphasizing the links between different dimensions. To be more precise, this thesis revolves around five central questions:

- What are the defect types?

- How are defects distributed in files?
- Do defects cluster in files?
- How to predict the hot files of defects?
- How do different projects compare in terms of defect characteristics?

1.3 Research goals

The goals of this thesis are:

- Develop effective techniques to address the research questions
- Implement the techniques in tools
- Evaluate the techniques and tools on sample projects
- Document the techniques, the design of the tools, and the results of evaluation

1.4 Research methodology

We develop new techniques to address the research questions in Section 1.2, and evaluate the proposed techniques on sample projects. We selected sample projects from Honeywell ACS to perform our study. The methodology is shown in Figure 1-2.

1.4.1 Data source

Experimental data for the selected sample projects are downloaded separately from the defect tracking system (JIRA [2]), the code review system (Crucible [3]), and the version control system (Subversion/Fisheye [4]). The downloaded files are in XML format.

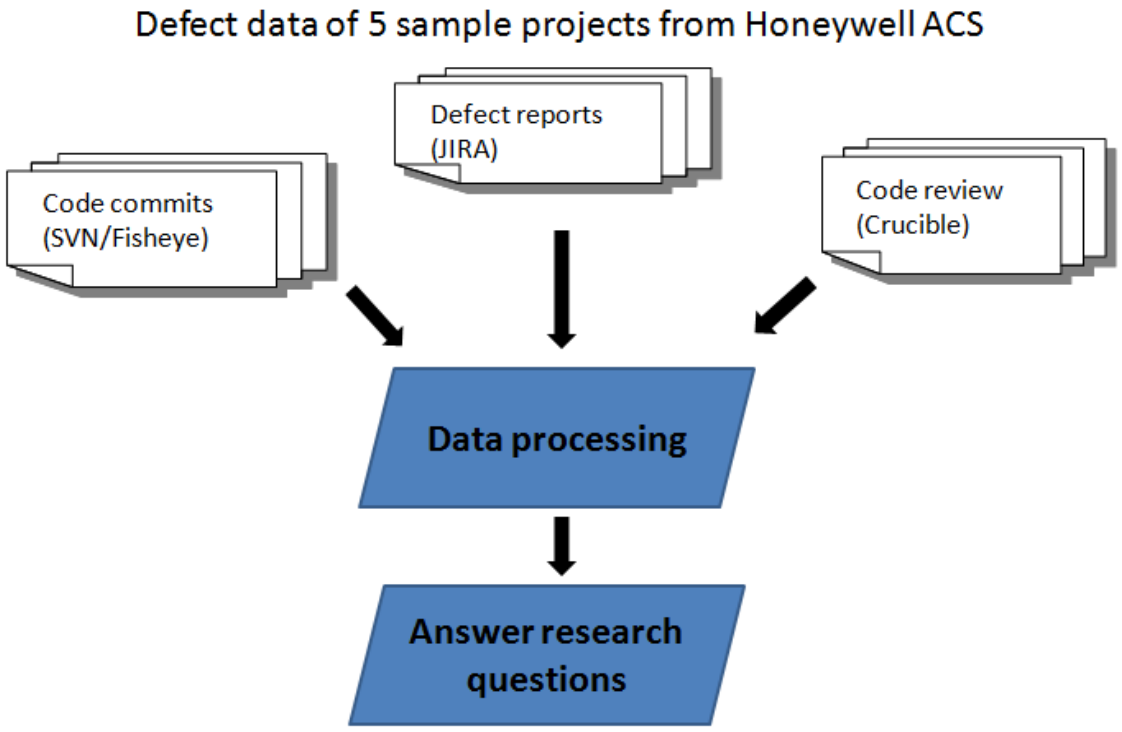


Figure 1-2: Research Methodology

1.4.2 Data processing

We processed the downloaded data using a command line tool for algorithm development and evaluation. We develop algorithms to perform defect data mining and answer the research questions in Section 1.2.

1.5 Contributions

This thesis makes the following contributions.

1.5.1 New techniques

Indexing defects by keywords

This thesis proposes to infer defect types by keywords. Before keyword extraction, defect reports are preprocessed to clean up the text, and some defect reports are

indexed manually as training samples to build the keyword extraction model. Then, a machine learning algorithm applies the trained models to the rest of the defect reports to extract keywords. After keyword extraction, a post-processing stage is applied to gather, filter and rank and select the keywords. The produced keywords are then used to index all the defect reports into groups.

Studying the properties of the indexed defects

This thesis proposes to study the properties of the indexed defects by leveraging code reviews and code commits. After inferring the types of defects by keywords, code reviews and code commits are sorted by defect types, and metrics of defect distribution and clustering are applied to answer research questions in section 1.2. A known defect prediction algorithm is adopted to predict the hot files of defects for the defect types of interest, with defect hit rate 50%-90%. Finally, sample projects are compared by their defect characteristics.

1.5.2 New tools

bugc: A command line tool for defect data integration

A command line tool, bugc, is developed in this thesis to process the defect data in XML format. The front-end of the tool is a XML parser [43]. An open-source keyword extraction tool, Kea [5, 6], is integrated with the tool to perform keyword indexing of defects by keywords. The pre-processing and post-processing stages of defect indexing, as well as metrics to study the properties of the indexed defects are implemented in this tool.

1.6 Thesis outline

This thesis is organized as follows:

Chapter 2 reviews existing approaches to defect data integration. Section 2.1 reviews existing approaches to studying defects and defect classification. Section 2.2

reviews existing approaches to integrating defect classification with code reviews. Section 2.3 reviews existing approaches to integrating defect classification with code commits. A summary is given in in Section 2.4.

Chapter 3 reviews the current practice of defect data integration at Honeywell. Section 3.1 reviews the defect dataset and the current practice of defect classification at Honeywell. Section 3.2 reviews the links between defect classification and code reviews. Section 3.3 reviews the links between defect classification and code commits. A summary is given in Section 3.4.

Chapter 4 describes the methodology of indexing defects by keywords. Section 4.1 explains the motivation to index defects by keywords. Section 4.2 gives an overview of automatic keyword extraction algorithms. Section 4.3 presents the required components for indexing defects by keywords. Section 4.4 describes the process of filtering text for keyword extraction. Section 4.5 describes the process of extracting raw keywords. Section 4.6 describes the process of collecting, filtering and ranking keywords. Section 4.7 introduces the algorithm of indexing defects by keywords. Finally, a summary is given in Section 4.8.

Chapter 5 describes the methodology of studying the properties of the indexed defects. Section 5.1 describes the methodology of inferring defect types by keywords. Section 5.2 describes the methodology to study the distribution of defects in files. Section 5.3 describes the methodology to study the clustering of defects in files. Section 5.4 describes the methodology to predict the hot files of defects. Finally, a summary is given in Section 5.5.

Chapter 6 evaluates of the proposed approach of defect data integration to two Honeywell ACS projects. Section 6.1 gives an outline of evaluation strategy. Section 6.2 describes the results of a JIRA server extention project. Section 6.3 describes the results of a gas ignition controller project. Section 6.4 discusses the implications of the results. Finally, a summary is given in Section 6.5.

Chapter 7 provides the overview of a defect data integration tool. Section 7.1 describes the Java implementation of this tool. Section 7.2 describes the examples of using the tool. Section 7.3 gives a list of the command line arguments. A summary

is given in Section 7.4.

Chapter 8 gives the conclusions in Section 8.1 and an outline of future work in Section 8.2.

Chapter 2

Prior work: State of the art

> Reliable and transparent programs are usually not in the interest of the designer.
>
> Niklaus Wirth

This chapter reviews existing approaches to study defect characteristics, and organizes the reviews into three sections:

- Reviews of defects and defect classification (Section 2.1)
- Reviews of integrating defect classification with code reviews (Section 2.2)
- Reviews of integrating defect classification with code commits (Section 2.3)

Each section covers one aspect of the defect dataset, but the approaches differ significantly. We identify the strengths and weaknesses of existing approaches and discuss in Section 2.4 why what are the limitations of them.

2.1 Defects vs. defect classification

Software system defects, commonly referred to as bugs, are always hot topics of computer science. Two groups of researchers are particularly active in the research of defects: the software engineering community and the computer systems community.

For the software engineering community, the number of defects is a direct measure of the quality of software development process, and the reduction of defect injection becomes the high-priority target of software engineering. For the computer systems community, on the other hand, defects pose threats to the performance, security and safety of computer systems. In multitier computer systems, many subsystems depend on others to function properly, and defects are serious issues for systems dependability. Note that both the software engineering community and the computer systems community use defect classification as an important tool to study defects.

Software engineering studies have focused on defect classification and software quality measurement. Chillarege et al. proposed Orthogonal Defect Classification (ODC) [7] to measure software development process. Some researchers focused on software quality measurement and modeling without really considering the classes of defects. Fenton and Ohlsson gave a quantitative analysis of the faults and failure in a complex software systems [8]. These studies mainly classify defects manually or address the measurement and modeling of defect counts without distinguishing the classes of defects explicitly.

Computer systems studies have focused on defect classification and systems behavior in the presence of defects. Some researchers used static program analysis to detect defects related to system runtime behavior such as race conditions and buffer management [9, 10]. Some researchers classify defects by manual reviews of error logs, memory dumps and other traces. Gray surveyed the defects in Tandem systems between 1985 and 1990 [11]. Sullivan and Chillarege compared the defects in database management systems and operating systems [12]. Lee and Iyer studied the defects and their symptoms in the Tandem Guardian90 operating system [13]. These studies classify defects either manually or automatically, and the classified defects are those related to system runtime behavior, such as pointers, interrupts, dead locks, buffer management, etc.

More recently, some researchers studied the inherent accuracy problems of static program analysis and proposed to use statistical methods to counter the overapproximation of static program analysis [14]. More static checkers were developed

[15, 16, 17, 18, 19] and the coupling of these checkers with statistical methods, such as ranking, user feedback and probabilistic inference were reported [20, 21, 22]. These studies used statistical methods to enlarge the scope of static program analysis to defect classes such as input and output interface, resource handling, etc.

2.2 Defect classification vs. code reviews

Code reviews have attracted the interest of the software engineering community. Kemerer and Paulk focused on incorporating code reviews into the modeling defect counts without explicitly distinguishing defect classes [23]. Mantyla and Lassenius argued what were the types of defects really discovered in code reviews [24], and they showed the types of defects mostly discovered in code reviews were defects unrelated to system runtime behavior. The process of identifying defects discovered by code reviews was entirely manual.

2.3 Defect classification vs. code commits

Code commits have attracted the interest of the software engineering community. Nagappan and Ball proposed to use code churn measures to predict system defect density [25]. Caching techniques to predict defect hot spots in source files have been reported [26]. A metric was proposed by Google to predict the hot spots of the defects [27].

Code commits have also attracted the attention of the computer systems community. The information contained in code commits includes many snapshots of source files, giving researchers opportunities to study defect types, defect lifetime, defect distribution, defect clustering, etc. Chou et al. presented an empirical study of software system defects found in operating systems using defects found by twelve static checkers, and studied the dominant defect types, the defect lifetime, the defect distribution and clustering for the Linux operating system. The methodology was automatic and a large part of this thesis was inspired by this it [28]. Li et al. presented a tool to find

copy-and-paste bugs [29]. Tan et al. studied methods to infer bugs from comments [30]. Yin et al. studied how bug fixes actually created more bugs [31]. These studies mostly used automatic tools to scan large codebases of computer systems.

2.4 Summary

Table 2.1: Research with different emphases on defect data

Research	Methods [a]	Defect types [b]	Reviews [c]	Commits [d]
Chillarege et al. [7]	manual	function, interface, etc.	yes	none
Mantyla and Lassenius [24]	manual	functional, evolvable	yes	none
Nagappan and Ball [25]	manual	none	none	yes
Chou et al. [28]	automatic	null, range, float, lock, etc.	none	yes
Tan et al. [30]	automatic	copy-and-paste	none	yes
Yin et al. [31]	automatic	fixes	none	yes

[a]Methods of defect finding: manual reviews or using automatic tools
[b]Defects are classified by types in research
[c]Code reviews are used in research
[d]Code commits are used in research

Prior research has focused on different aspects of defect dataset, and some are summarized in Table 2.1. Three software engineering studies and three computer systems studies are highlighted, with different emphases on defect data. The approaches taken by software engineering studies are quite different from those of computer systems studies: 1) Software engineering studies mostly use manual reviews to review defects, whereas computer systems studies use automatic tools to scan large codebases. 2) Some software engineering studies model defect counts without explicitly distinguishing defect types, whereas computer systems studies usually relate to the root causes of defects. 3) Some software engineering studies model the impacts of code reviews, while computer systems studies mostly discard code reviews. This could be explained by the unavailability of code reviews data in most open-source computer systems.

Limitations of the past research are clear when comparing software engineering studies with those of computer systems studies: 1) Manual reviews are not scalable,

whereas analysis using automatic tools can scale to large dataset. Also manual reviews require experienced individuals, whereas automatic tools need not. 2) Defect types are useful as they provide the physical origins of defects, and they are are very helpful for defect analysis. But defect reports are usually not classified by their physical origins, and program analysis tools only operate on source code. Defects in the defect reports cover defects from programming errors, performance issues, high-level design to user interface, a more comprehensive variety than defects covered by previous studies using static program analysis. 3) The availability of dataset is critical for defect analysis, and code reviews data is usually not available in wide practice.

Chapter 3

Prior work: State of the practice at Honeywell

> It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
>
> Arthur Conan Doyle, A Scandal in Bohemia

This chapter reviews the current practice of defect data integration at Honeywell ACS, and organizes the reviews into three sections:

- Reviews of defects and defect classification (Section 3.1)
- Reviews of integrating defect classification with code reviews (Section 3.2)
- Reviews of integrating defect classification with code commits (Section 3.3)

Each section covers one aspect of the defect dataset at Honeywell ACS. We identify the weaknesses of each group and outline directions of improvement in Section 3.4 by comparing the practice at Honeywell to the state of the art methods in Chapter 2.

3.1 Defects vs. defect classification

Defect reports are collected and stored in a defect tracking system (JIRA [2]), and code reviews and code commits are linked to defect reports by defect keys. Nevertheless, the links are not well-established for all projects. The statistics of eight Honeywell ACS projects, as of May 2013, are as shown in Table 3.1.

Table 3.1: Defect data for eight sample projects

Project	Defects	Commits	Reviews
Project A	450	3901	62
Project B	616	2982	11
Project C	355	3666	0
Project D	194	1238	0
Project E	131	2415	0
Project F	408	0	0
Project G	670	0	0
Project H	793	0	0
Total	3617	14202	73

Two things are worth noting. 1) With 8/1979 projects of Honeywell ACS, the defect dataset is already too large to carry out manual reviews, and a forward projection to all Honeywell ACS projects makes manual reviews impractical. An effective approach to defect analysis and prediction must be based on automatic tools. 2) Defect data is incomplete, only a few projects have the three-dimensional data, with code reviews data the most incomplete. An effective approach would be to drop cod reviews and focus on the integration of defect classification and cod commits.

Defect classification is an important tool for the research of defects [7, 24, 28, 30, 31]. The current practice of defect classification at Honeywell ACS is ODC-like [7, 32]. An optional field in a defect report provides the optional choice to specify a defect type. Nine predefined defect types are shown in Table 3.2.

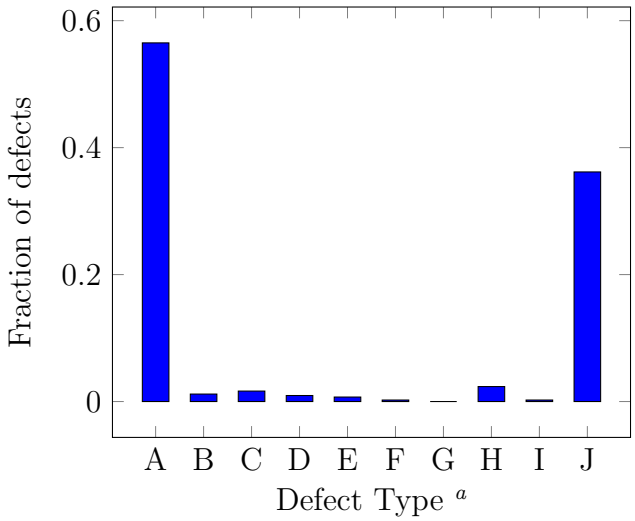
Table 3.2: Predefined defect types

Defect types
SW-Implementation
SW-Design
Test-Procedure
Requirements
Customer-Documentation
HW-Design
Others
None
N/A

There are three problems in this defect classification process. 1) The defect type field is optional, and many defect reports just leave this field unspecified. Also three types "Others," "None," and "N/A" essentially provide no information. 2) The predefined defect types are ODC-like [7] and are more suitable for process measurement rather than defect analysis. The predefined defect types have no direct relations to the root causes of defects as in the previous studies [28, 30, 31]. 3) The predefined defect types are too broad. For example, too many defects could be put in the category of "SW-Implementation," making the defect classification ambiguous. For example, the defect type distribution of project A in Figure 3-1, the defects put in the defect type of **SW-Implementation** almost cover more than half of the total defects, and around 30% of the defects are not specified with defect types. Thus defect analysis like the previous study [28] is very difficult.

3.2 Defect classification vs. code reviews

The current practice of integrating defect classification and code reviews at Honeywell ACS has three problems. 1) The current practice of code reviews is limited, resulting in very limited data. The author only found two projects with code reviews, with 73 code reviews in total. This makes any statistical analysis inadequate because of the very limited sample space. 2) Out of the already limited code reviews, only a few of them target on defects. For example, only 1 out of 62 code reviews of Project A



[a]A: SW-Implementation, B: SW-Design, C: Test-Procedure, D: Requirements, E: Customer-Documentation, F: HW-Design, G: Others, H: None, I: N/A, J: Unspecified

Figure 3-1: Defect type distribution of Project A

focused on a defect related to the layout of the user interface, and other 61 reviews were not related to defects. 3) As mentioned in Section 3.1, defect classification is too ambiguous to relate defects to their root causes. With limited data, integration like the prior research [24] is difficult.

Perhaps one thing that could be learned was that defects were not reviewed enough. Especially for defects unrelated to system runtime behavior [24], they should be reviewed more to prevent similar defects from happening. But this observation is obvious from the viewpoint of software engineering, and we avoid addressing this in the following chapters. It is for this reason that we drop code reviews in the following chapters to focus on the integration of defect classification and code commits.

3.3 Defect classification vs. code commits

The current practice of integrating defect classification and code commits at Honeywell ACS has two problems. 1) The current practice of linking code commits to defect reports is not universally practiced, resulting in limited projects with enough data. The author only found five projects with enough connections between defect

reports and code commits. 2) As mentioned in Section 3.1, defect classification is too ambiguous to relate defects to their root causes, making integration like the prior research [28] difficult. We address this difficulty and explore alternative solutions in Chapter 4 and Chapter 5.

3.4 Summary

Table 3.3: Defect data integration at Honeywell ACS

	Defect reports	Code reviews	Code commits
Defect classification	Optional field. Predefined types too broad.	Not enough data. Classification ambiguous.	Some data. Classification ambiguous.

The current problems of integrating defect reports, code reviews and code commits at Honeywell ACS are summarized in Table 3.3. We highlight defect classification in each column to emphasize it is the key to defect data integration, and it is also the bottleneck of the current practice of defect data integration at Honeywell ACS. We present the solutions to this problem in Chapter 4 and Chapter 5. The conceptual process of solving the defect data integration problem at Honeywell is illustrated in Figure 3-2, starting from identifying the bottleneck of the problem, proposing a new solution, looking into available data, to the solutions of the problem.

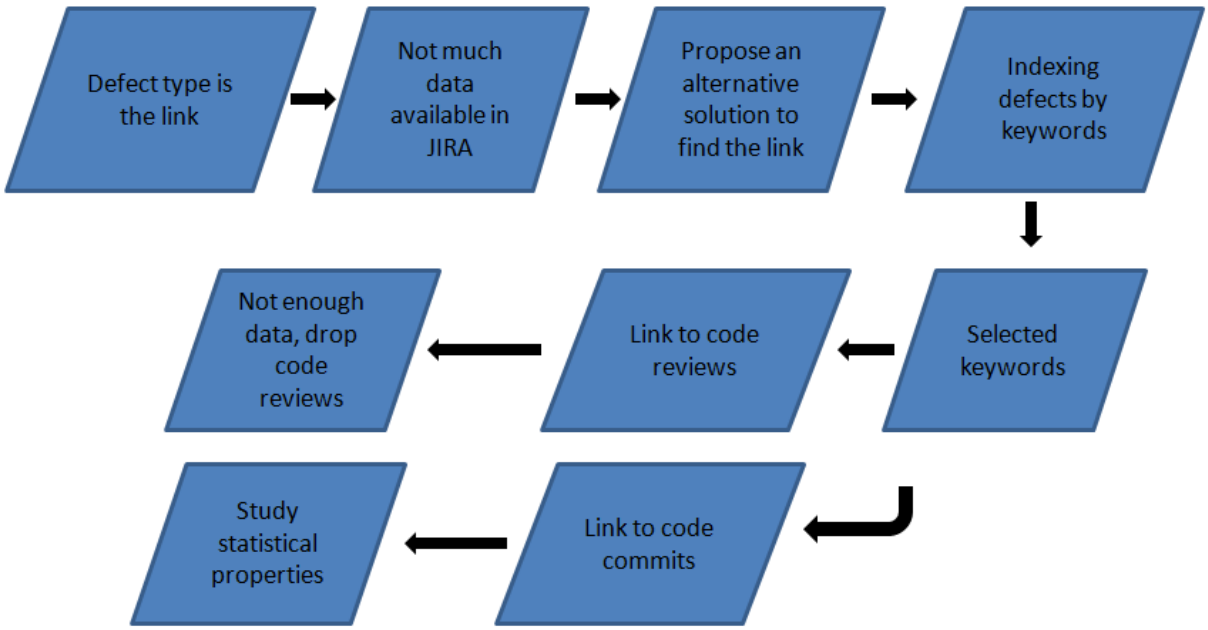


Figure 3-2: The conceptual process of solving the defect data integration problem at Honeywell ACS

Chapter 4

Methodology of indexing defects by keywords

> Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise.
>
> John Tukey

This thesis presents a new approach to defect data integration. Chapter 3 identified the bottleneck of defect data integration: defect classification. This chapter introduces a new technique, defect indexing by keywords, to solve this problem. It focuses on the motivation of this new technique, the background and latest development of automatic keyword extraction, and the methodology of applying this technique to defect classification. Section 4.1 explains the motivation to index defects by keywords. Section 4.2 gives an overview of automatic keyword extraction algorithms. Section 4.3 presents the required components of indexing defects by keywords. Section 4.4 describes the process of filtering raw defect reports. Section 4.5 describes the process of extracting raw keywords. Section 4.6 describes the process of collecting, filtering and ranking keywords. Section 4.7 describes the algorithm of indexing

defects by keywords. Finally, a summary is given in Section 4.8.

4.1 Motivation

The idea that computers can understand the natural languages in defect reports is impractical. Topics related to natural language processing have been researched for many decades, but rarely achieved a level comparable to human performance. Researchers must deal with the complexity and highly subjective nature of natural languages. In many cases, judging which machine understanding is more accurate is difficult.

Indexing defects by keywords, on the other hand, is easier. The question "What are the keywords in the defect reports?" does assume some understanding of natural languages, but restricts the answer to some keywords that describe the report's main topics. Deep understanding is not necessary, since even human indexers skim through the report to spot the keywords. Indexing defects by keywords is also easier to evaluate. Those keywords that the experienced developers agree must be the right ones. Over the years, automatic keyword extraction algorithms have made progress and have become more and more human-competitive.

In light of the progress of automatic keyword extraction techniques, we would like to explore indexing as our approach to infer defect types from the keywords found in the descriptions of defect reports. Figure 4-2 is a defect report on string handling problems, with the keyword **string** highlighted. The defect report is in XML format and contains many fields, including **summary**, **description**, **comment**, etc. The keyword **string** was spot by a programmer, with some knowledge of Java programming. This defect report referred to an exception caused by the evaluation of a type mismatch expression between type **long** and type **string**. The choice of **string** as a keyword was a matter of personal taste, since other choices like **long** or **== (equality test)** were also possible. Defect reports with the keyword **string** can be grouped together for further study.

Indexing defects by keywords is our approach to classifying defects 4-1. We group

defects with common keywords and study their properties. In particular, we would like to answer the research questions in Section 1.2. One might argue that the defects indexed by a keyword not necessarily share the same root cause. For example, the defect reports indexed by the keyword **string** might not all related to **string** handling problems. We understand keywords are only an approximation of the meaning of the text, and this thesis explores the approximation power of this approach.

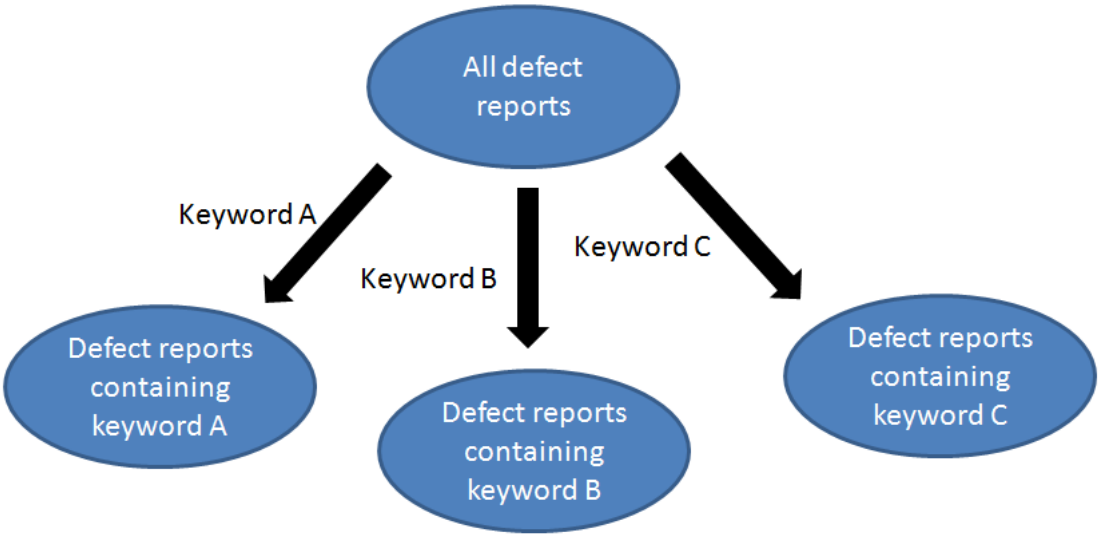


Figure 4-1: Defect classification by keywords

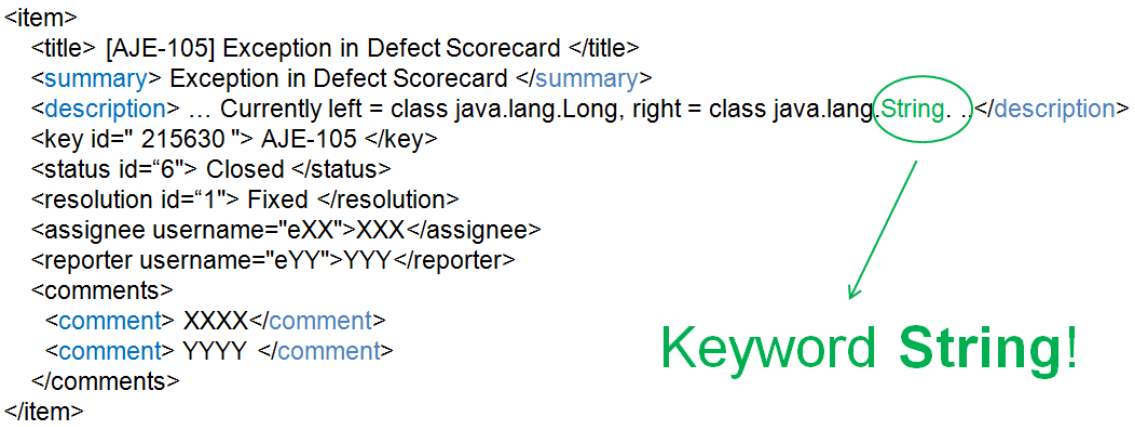


Figure 4-2: An example of defect report

The motivating example in Figure 4-2 illustrates the challenges of indexing defects by keywords: 1) Random words should never be marked as keywords. Before keyword extraction, filtering the raw text is necessary in order to reduce as much irrelevant text as possible (Section 4.4). 2) Extracting good keywords from the defect reports requires some background knowledge of the defects, making a supervised machine learning approach to keyword extraction appealing. The keyword extraction model should be trained by sample reports indexed by programmers (Section 4.5). 3) Some defect reports are badly written that it would be difficult to spot meaningful keywords because the words contained in the defect reports are misleading or unrelated to the root causes of the defects. Therefore, the extracted keywords from different defect reports should be collected and ranked by programmers to find the most representative keywords (Section 4.6). 4) Multiple keywords from the same report should be compared in order to determine the most representative one, so metrics to compare the importance of keywords are necessary (Section 4.7).

4.2 Overview of automatic keyword extraction

Automatic keyword extraction strategies differ from each other depending on the origins of topics: a controlled vocabulary, document text or terms assigned to similar documents. Medelyan [1] organizes the large number of published methods for automatic topic indexing in three major groups: 1) Term assignment methods, which use a controlled vocabulary 2) Keyword extraction methods, which derive topics from document text 3) Tagging methods, which mine topics from any possible source. For indexing of defects by keywords, there is no controlled vocabulary and no terms assigned to similar documents. So we focus on keyword extraction methods in this thesis.

Medelyan [1] summarized the keyword extraction in two stages: candidate generation and filtering. Methods for candidate generation vary from n-gram extraction [33, 34, 35] to shallow parsing [36]. N-gram extraction often results in ungrammatical phrases. Parsing considers part-of-speech information and is more accurate, however

it is only available in some languages. Methods of filtering can be grouped in two: supervised methods based on machine learning and heuristic methods. For indexing defects by keywords, model training is necessary so we discard heuristic methods. One popular supervised technique is proposed by Witten. Witten et al. [34] developed Kea, the Keyphrase Extraction Algorithm. Kea is the algorithm adopted in this thesis, and we discuss it in greater details.

Kea performs three steps for generating candidates. 1) Kea determines textual sequences defined by orthographical boundaries such as punctuation marks, numbers, and newlines, and then these sequences are split into tokens. 2) Kea extracts candidate phrases that consist of one or more words that do not begin or end with a stopword. 3) each candidate is stemmed using the iterated Lovins [37] stemmer and the most frequent version is saved.

Kea performs four steps for filtering. 1) Kea computes two features for each keyphrase candidate: the TF · IDF [1] measure and the position of the first occurrence [2]. 2) Kea uses a Naive Bayes classifier [3] [38] to analyze training data and creates two sets of weights for both candidates that match manually assigned keyphrases and for all other candidates. 3) Kea calculates the overall probability of each candidate being a keyphrase based on these weights. 4) Kea ranks the candidates according to their probabilities, and the top ranked phrases are included into the results.

The flow chart of automatic keyword extraction are shown in Figure 4-3 [1]. Kea [6] is now integrated in an open-source machine learning software, Weka [39]. Im-

[1] TF · IDF score measures how specific a phrase P is to a given document D:
TF · IDF(P, D) = Pr[phrase P in D] · -log Pr[P in a document].
The first probability is estimated by the number of times the phrase P occurs in the document D, and the second one by the number of documents in the training group that contain P (excluding D) [34]

[2] The distance of a phrase from the beginning of a document is the number of words that precede its first appearance. It is normalized by the number of words in the document, so that the result is a number between 0 and 1. [34]

[3] The naive Bayes learning scheme assumes that TF · IDF and distance are independent. The probability that a phrase is a keyphrase given that it has discretized TF · IDF value T and discretized distance D is:
Pr[key | T, D] = (Pr[T | key] · Pr[D | key] · Pr[key]) $\Big/$ Pr[T | D],
where Pr[T | key] is the probability that a keyphrase has TF · IDF score T, Pr[D | key] is the probability that it has distance D, Pr[key] is the *a priori* probability that a phrase is a keyphrase, and Pr[T | D] is a factor for normalizing Pr[key | T, D] a number between 0 and 1. [34]

provements of Kea have been reported [5, 40, 36, 41, 42], but they are outside the scope of this thesis.

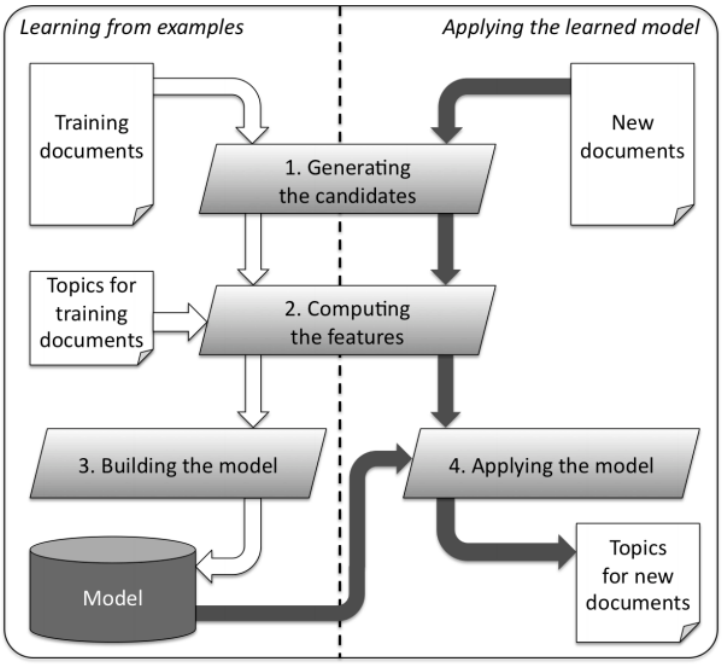


Figure 4-3: Flow chart of automatic keyphrase extraction [1]

4.3 Components

The flow of indexing defects by keywords contains four stages.

- Text filtering (Section 4.4)
- Keyword generation (Section 4.5)
- Keyword selection (Section 4.6)
- Indexing defects by keywords (Section 4.7)

The four stages are illustrated in the flow chart in Figure 4-4. 1) Filtering of defect reports is done before model training and keyword extraction. The defect reports contain many text fields and we only select some of them for keyword extraction. The

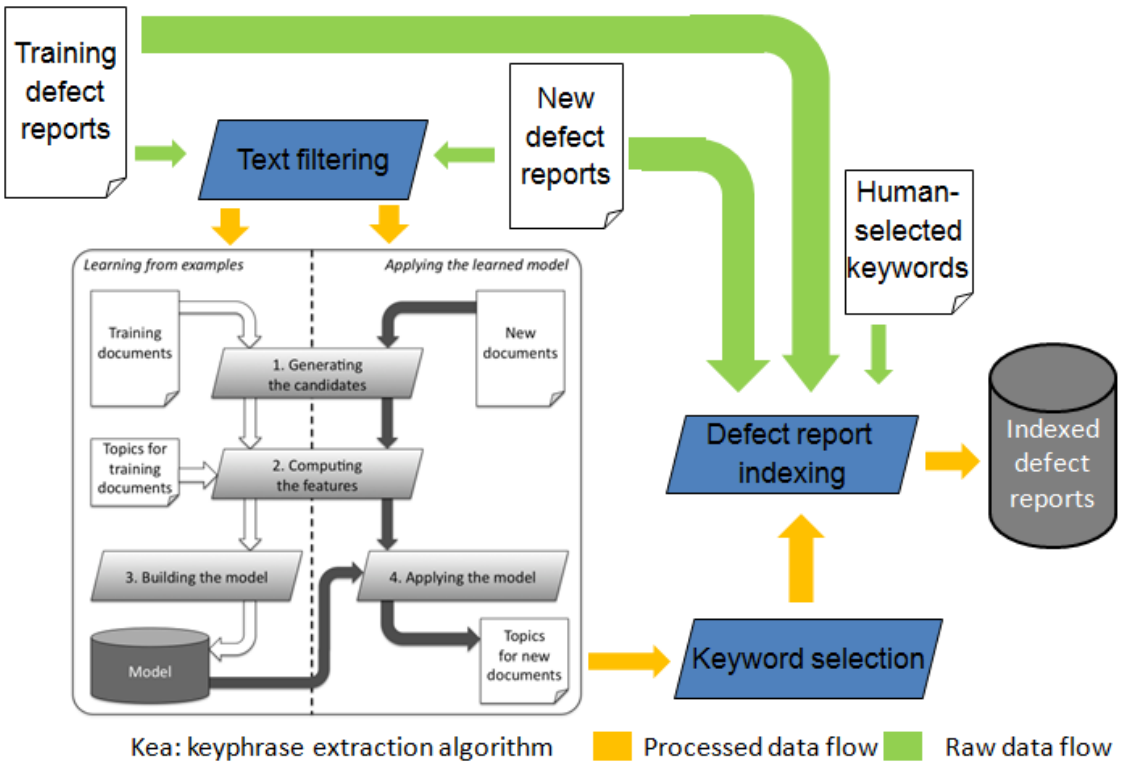


Figure 4-4: Flow chart of indexing defects by keywords

selected fields are configurable through a text file. We selected **summary**, **description** and **comments** in this thesis. The procedure is described by Algorithm 1. The direction of data flow is illustrated in Figure 4-4 by the green arrows. 2) Keyword generation using Kea and Weka includes processing the filtered defect reports, building the keyword extraction model, and extracting the desired keywords. The procedure is described by Algorithm 2. This part is illustrated by the Kea block in Figure 4-4. 3) Keyword selection by collecting the generated keywords into a list to be selected by human indexers. The procedure is described by Algorithm 3. This part is illustrated by the processed data flow in orange arrows in Figure 4-4. 4) Indexing the defect reports by the selected keywords, the priority of keywords are determined by the *FrequencyScore* and *FirstPositionScore*. The procedure is described by Algorithm 4. This part is shown as the last stage in Figure 4-4, with the indexed defect reports as the output.

4.4 Text filtering

Text filtering filters raw defect reports as the example in Figure 4-7 to clear up the textual information and produces reports as the example in Figure 4-8. Text filtering is configured by a defect pattern definition file as the examples in Figure 4-5, 4-6. Depending on the granularity of filtering, three filtering schemes are implemented: 1) Filtering by fields as defined by the example in Figure 4-5, 2) filtering by keywords as defined by the example in Figure 4-6, and 3) filtering the irrelevant textual information, such as the **&gt** symbol. The procedure is described by Algorithm 1.

```
<!--
Pattern 1:
        search for keywords: memory
-->
<pattern>
  <name>Memory defects</name>
  <description>
    <match>
  </description>
  <summary>
    <match>
  </summary>
  <comments>
    <match>
  </comments>
</pattern>
```

Figure 4-5: A defect pattern without keyword matching fields

Algorithm 1 Text filtering of defect reports
Input: a list of defect reports and a configuration file
Output: a list of filtered defect reports

Require: x is a list of defect reports, y is a configuration file
1: **function** TextFilteringOfDefectReports(x, y)
2: **for** $x_i : x$ **do**
3: **if** x_i has fields specified in y **then**
4: $x_i \leftarrow$ FilterEachField(x_i, y)
5: $z_i \leftarrow$ OutputFieldsInOrderWithFieldnames(x_i)
6: **return** z ▷ z is a list of filtered defect reports

```
<!--
Pattern 1:
        search for keywords: memory
-->
<pattern>
  <name>Memory defects</name>
  <description>
    <match>
      <name>description</name>
      <keywords>
        <keyword>memory</keyword>
      </keywords>
    </match>
  </description>
  <summary>
    <match>
      <name>summary</name>
      <keywords>
        <keyword>memory</keyword>
      </keywords>
    </match>
  </summary>
  <comments>
    <match>
      <name>comment</name>
      <keywords>
        <keyword>memory</keyword>
      </keywords>
    </match>
  </comments>
</pattern>
```

Figure 4-6: A defect pattern with keyword matching fields

4.5 Keyword generation

Keyword generation takes some filtered defect reports as the examples in Figure 4-8, with human indexed keywords in Table 4.1 and feed them to Kea [6] to build a keyword extraction model, and then apply this model to extract keywords of other filtered reports without human indexed keywords as the example in Figure 4-9. The input and output files are regulated by Kea and are organized in plain text format with specific filename extensions. The machine extracted keywords for the defect report in Figure 4-9 is shown in Table 4.2. The procedure is described by Algorithm 2.

```
<title>[GIC-574] Bad responce to HVAC Counter</title>
<link>https://acsjira.honeywell.com/browse/GIC-574</link>
<project id="10222" key="GIC">Gas Ignition Controls </project>
<description>&lt;p&gt;When device recieve HVAC Counter Query with defined counter return Bad counter.&lt;br/&gt;
When device recieve HVAC Counter Query with FF FF return list of all counters&lt;/p&gt;</description>
<environment></environment>
<key id="213599">GIC-574</key>
<summary>Bad responce to HVAC Counter</summary>
<type id="1" iconUrl="https://acsjira.honeywell.com/images/icons/issuetypes/bug.png">Defect</type>
<priority id="10" iconUrl="https://acsjira.honeywell.com/images/icons/priorities/trivial.png">None</priority>
<status id="6" iconUrl="https://acsjira.honeywell.com/images/icons/statuses/closed.png">Closed</status>
<resolution id="2">Won&apos;t Fix</resolution>
<assignee username="e353066">Zdenek Orsag</assignee>
<reporter username="e353066">Zdenek Orsag</reporter>
<labels></labels>
<created>Tue, 15 Mar 2011 12:46:05 +0100</created>
<updated>Fri, 1 Jul 2011 07:00:13 +0200</updated>
<resolved>Wed, 16 Mar 2011 12:44:10 +0100</resolved>
<due></due>
<votes>0</votes>
<watches>0</watches>
<comments>
  <comment id="410847" author="e353066" created="Fri, 1 Jul 2011 07:00:13 +0200"  >&lt;p&gt;Specification updated so this is not issue anymore&lt;/p&gt
  <comment id="350359" author="e461459" created="Mon, 28 Mar 2011 13:44:41 +0200"  >&lt;p&gt;Requirement PRS-DSI-IFDC-2.3.2.8-4 has been updated accord
  <comment id="345451" author="e461459" created="Wed, 16 Mar 2011 12:44:10 +0100"  >&lt;p&gt;Due to memory constraints it is not possible to retrieve
</comments>
```


Figure 4-7: A raw defect report in XML format

```
Description:
 When device recieve HVAC Counter Query with defined counter return Bad counter.
When device recieve HVAC Counter Query with FF FF return list of all counters

Summary:
Bad responce to HVAC Counter

Comments:
 Specification updated so this is not issue anymore
 Requirement PRS-DSI-IFDC-2.3.2.8-4 has been updated accordingly.
 Due to memory constraints it is not possible to retrieve value of single counter. Implemented is only support of wildcard query (0xFFFF) when values of
```

Figure 4-8: A filtered defect report

4.6 Keyword selection

Keyword selection is done by human indexers to choose some representative keywords from the machine extracted keywords as the example in Table 4.2. In this process, some meaningless or not so representative keywords are dropped. For example, we see only **heating** would be a more meaningful keyword in Table 4.2. The procedure is described by Algorithm 3.

Algorithm 2 Keyword generation
Input: two lists of defect reports (one for model training, one for machine indexing), a list of keyword files indexed by human indexers
Output: a list of keyword files

Require: x is a list of filtered defect reports for model training, y is a list of human-indexed keyword files for x, z is a list of filtered defect reports for machine indexing

```
1: function KeywordGeneration(x, y, z)
2:     Put x and y in the same directory
3:     Configure Kea in the training mode of free indexing with necessary parameters
4:     m ← TrainKeywordExtractionModel(x, y)
5:     for z_i : z do
6:         k ← ApplyKeywordExtractionModel(z_i, m)
7:         Output(k)                                  ▷ write keywords to a file
```

Algorithm 3 Keyword selection
Input: a list of keyword files
Output: a keyword file

Require: u is a list of keyword files

```
1: function KeywordSelection(u)
2:     for u_i : u do
3:         k ← CollectKeywords(u_i)
4:         k ← FilterRepeatedKeywords(k)
5:         k ← RankKeywordsByFrequency(k)
6:     v ← Output(k)
7:     w ← HumanSelectMeaningfulKeywords(v)
8:     return w                                       ▷ w is a keyword file
```

Table 4.1: An example of human indexed keywords

HVAC Counter Query
counter
memory

```
Description:
 On 90+ model when is closed HPS and W1 request device report fault 55 and h
In PRS is this valid only for 80+model

 SVN2956

Summary:
Device report h and 55 on 90+model

Comments:
 If HPS is closed prior to the heating state and heat request is set:

     80+ model: Fault 55 (level 2) is reported but heating is not blocked.
     90+ model: Fault 55 (level 2) is reported and heating is blocked. IFC is waiting for PS open.



 Behavior according to the PRS, chap. 6.1.2
 Reporter changed from Zdenek Orsag to Robert Pasz as Zdenek is no longer involved in this project.
   To follow Specification  - selected first solution.
  The cause was found. Two solution are proposed:

     Simple solution:

         Add a detection of model AFUE.


     Complex solution:

         Add detection of current state.
```

Figure 4-9: A filtered defect report in plain text format

4.7 Indexing defects by keywords

After keywords are selected, defect patterns as the example in Figure 4-6 are used to index defects by keywords. The keywords are case-insensitive. Note that defect patterns are used both for text filtering and defect indexing. The results of defect indexing are stored in a map data structure as shown in the example in Figure 4-10, and output in plain text format. The map data structure is designed to match the goal of classifying defects by keywords (Figure 4-1). Table 4.3 shows an example of the defects indexed by the keyword **memory**. A manual review showed that 7/10 of these defects were indeed related to memory management errors. The procedure is described by Algorithm 4. Equation 4.1 is used to calculate *FrequencyScore* and

Table 4.2: An example of machine indexed keywords

IFC
model
heating
report
Description
Closed HPS
HPS
Closed

FirstPositionScore, which are used to determine the dominant keyword when multiple keywords in different patterns are found in the same report (Figure 4-11). As shown in Algorithm 4, the procedure of determining the dominant keyword is first by comparing *FrequencyScore* for the maximum and then by comparing *FirstPosition-Score* for the minimum when *FrequencyScore*-s are equal. This procedure is simple and generalization of Equation 4.1 to multiple keywords in each pattern is presented in Section 5.1.

$$FrequencyScore(k, d) = \text{number of times k found in d}$$
$$FirstPositionScore(k, d) = \text{first occurence position of k in d} \tag{4.1}$$

where

the first occurence position is measured by the number of characters

4.8 Summary

This chapter presented the methodology of indexing defects by keywords in four stages: text filtering, keyword generation, keyword selection and indexing defects by keywords. The challenge of text filtering is to identify the unwanted text to be

Algorithm 4 Indexing defects by keywords
Input: a list of defect reports and a list of keywords
Output: a map of index keys with the associated list of defect reports

Require: x is a list of filtered defect reports, w is a list of keywords

```
1:  function IndexingDefectsByKeywords(x, w)
2:      z ← new Map for storing indexed defects
3:      for x_i : x do
4:          maxFreq ← 0
5:          leastFirstPos ← -1
6:          indexKey ← ""
7:          for w_j : w do
8:              if x_i contains w_j then
9:                  fs ← ComputeFrequencyScore(w_j, x_i)
10:                 fps ← ComputeFirstPositionScore(w_j, x_i)
11:                 if maxFreq ≤ fs then          ▷ determine the dominant keyword
12:                     maxFreq ← fps
13:                     indexKey ← w_j
14:                     leastFirstPos ← fps
15:                 else if fs = maxFreq then
16:                     if fps ≤ leastFirstPos then
17:                         indexKey ← w_j
18:                         leastFirstPos ← fps
19:         if indexKey ≠ "" then
20:             z ← z.put(indexKey, x_i)
21:     return z          ▷ z is a map of keywords with the associated list of defects
```

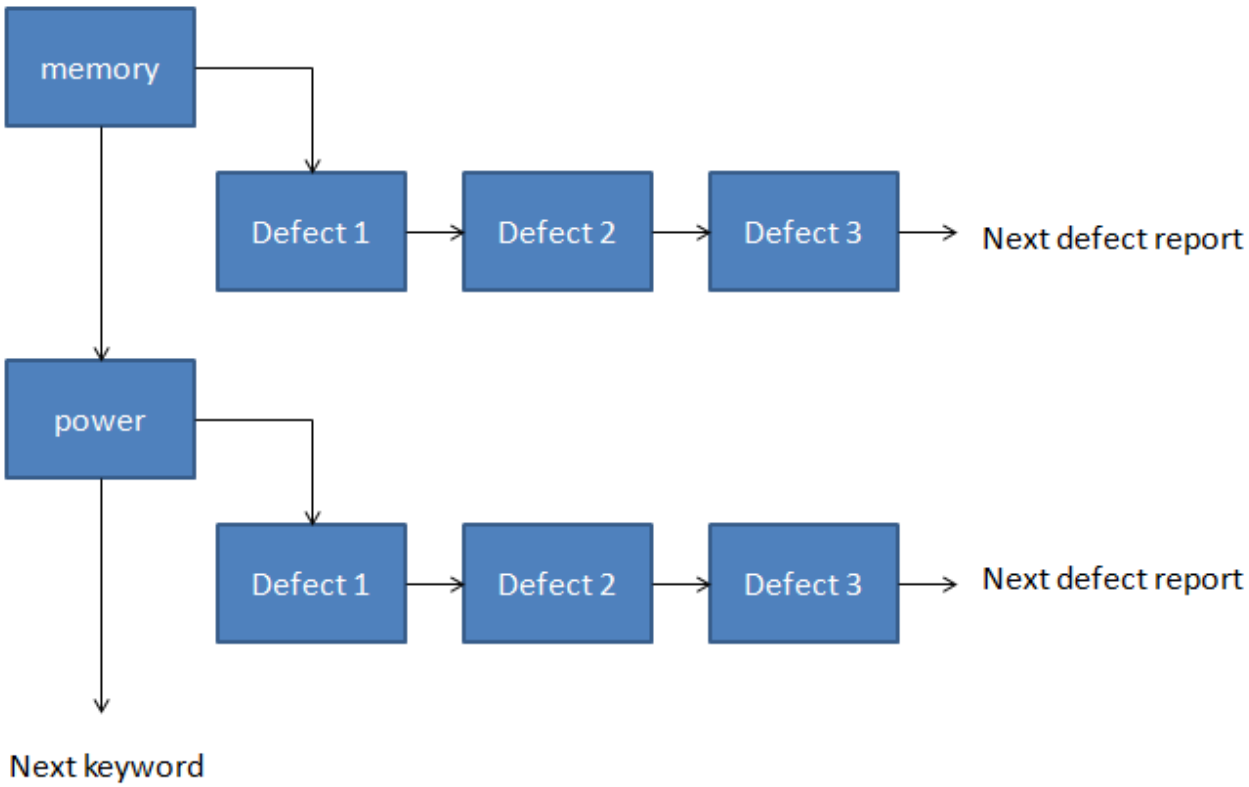


Figure 4-10: Map data structure for indexed defects

filtered. Keyword generation is performed by Kea, a tool integrated with bugc to extract keywords from filtered defect reports. Keyword selection in carried out by collecting keywords from files, ranking keywords by frequency, and human selection of keywords. The selected keywords are used to compose defect patterns. Indexing defects by keywords is accomplished by matching defect patterns on defect reports. Multiple matched keywords in different defect patterns are resolved by determining the dominant keyword.

Once defects are indexed into groups, we can study the properties of the defects in each group, in particular answering the research questions in Section 1.2. Chapter 5 explains how this can be done.

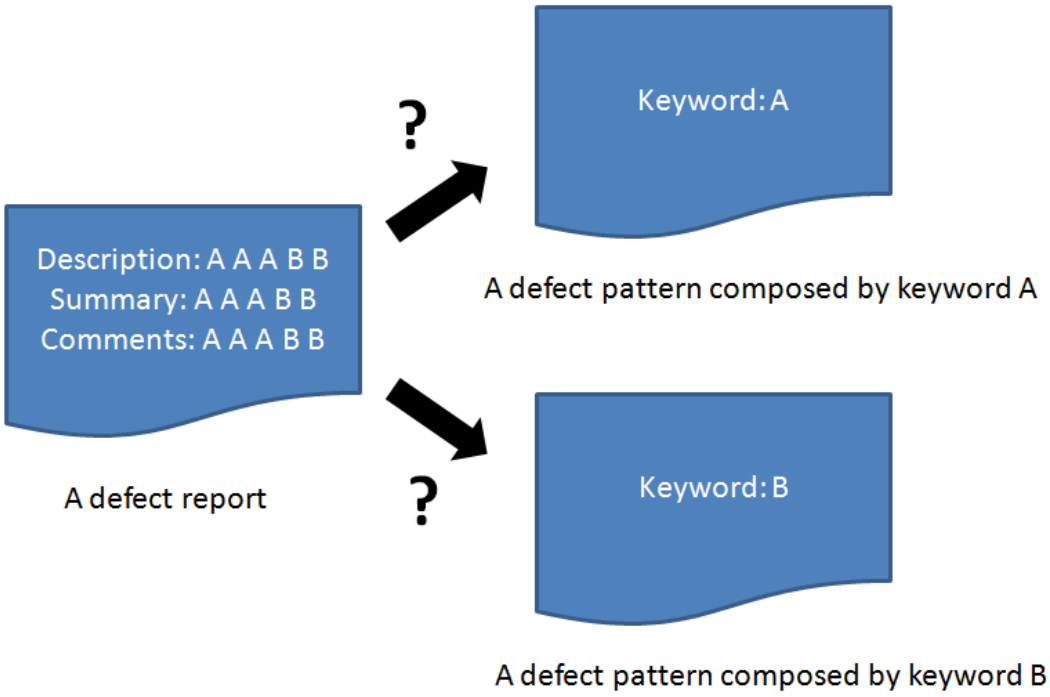


Figure 4-11: A defect report matched by multiple defect patterns composed by different keywords

Table 4.3: An example of defects indexed by the keyword **memory**

Defects indexed by keywords	
Keyword	Defect summary
memory	Bad responce to HVAC Counter
	Rheem Furnace Model and Serial Number are cleared after restart
	ECM Blower problem after User menu update
	no D4 alarm when memory card is invalid and data on cpu are valid
	d1 alarm active even if memory card with correct data connected
	Rheem Furnace Model and Serial Number are cleared after restart
	Overflow of array index
	ESD Failure of Memory Card Connector
	RAM usage
	Bad SSD display when is fault clearing

Chapter 5

Methodology of integrating defect indexing with code commits

Divide and conquer.

Julius Caesar

Chapter 4 introduced the techniques of indexing defects by keywords as a solution to defect classification. Once defects are classified into groups, we can study the properties of each group by leveraging code commits data (Figure 5-1). This chapter details the methodology to join defect indexing with code commits to answer the research questions in Section 1.2. Section 5.1 describes the methodology of inferring defect types by keywords. Section 5.2 describes the methodology to study the distribution of defects in files. Section 5.3 describes the methodology to study the clustering of defects in files. Section 5.4 describes the methodology to predict the hot files of defects. Finally, a summary is given in Section 5.5.

5.1 What are the defect types?

The idea of using keywords to represent defect types was introduced in Chapter 4. But sometimes indexing defects by defect patterns each composed by a single keyword is not enough. In this section, generalization of the previous indexing techniques is

developed. The procedure is described by Algorithm 5, with the generalization of Equation 4.1 in Equation 5.1. Note that the use of multiple keywords to compose defect patterns (Figure 5-2 and Figure 5-3) allows the recursive construction of defect classification trees (Figure 5-4), with the precision of classification refined in each level going down from the root to the leaves. Matching of multiple keywords uses AND logic: a defect report is matched to a defect pattern if all the specified keywords are found in it. An example of the fraction of the indexed defects is shown in Table 5.1, and examples of the indexed defects are shown in Table 5.2 and Table 5.3. Discussion of experimental results is presented in Chapter 6.

Algorithm 5 Inferring defect types by keywords
Input: a list of defect reports and a list of defect patterns
Output: a map of defect types with the associated list of defect reports

Require: x is a list of filtered defect reports, p is a list of defect patterns

```
1:  function InferringDefectTypesByKeywords(x, p)
2:      z ← new Map for storing indexed defects
3:      for x_i : x do
4:          maxFreq ← 0
5:          leastFirstPos ← -1
6:          defectType ← null
7:          for p_j : p do
8:              if ContainAllKeywords(x_i, p_j) then
9:                  fs ← ComputeGeneralizedFrequencyScore(p_j, x_i)
10:                 fps ← ComputeGeneralizedFirstPositionScore(p_j, x_i)
11:                 if maxFreq ≤ fs then          ▷ determine the dominant pattern
12:                     maxFreq ← fps
13:                     defectType ← p_j
14:                     leastFirstPos ← fps
15:                 else if fs = maxFreq then
16:                     if fps ≤ leastFirstPos then
17:                         defectType ← p_j
18:                         leastFirstPos ← fps
19:         if defectType ≠ null then
20:             z ← z.put(defectType, x_i)
21:     return z ▷ z is a map of defect types with the associated list of defect reports
```

p = p(k_1, k_2, ..., k_n)defect pattern with n keywords

$$GeneralizedFrequencyScore(p, d) = \sum_{i=1}^{n} \text{number of times } k_i \text{ found in d}$$

$$GeneralizedFirstPositionScore(p, d) = Min(\text{first occurence position of } k_i \text{ in d}) \tag{5.1}$$

where

the first occurence position is measured by the number of characters

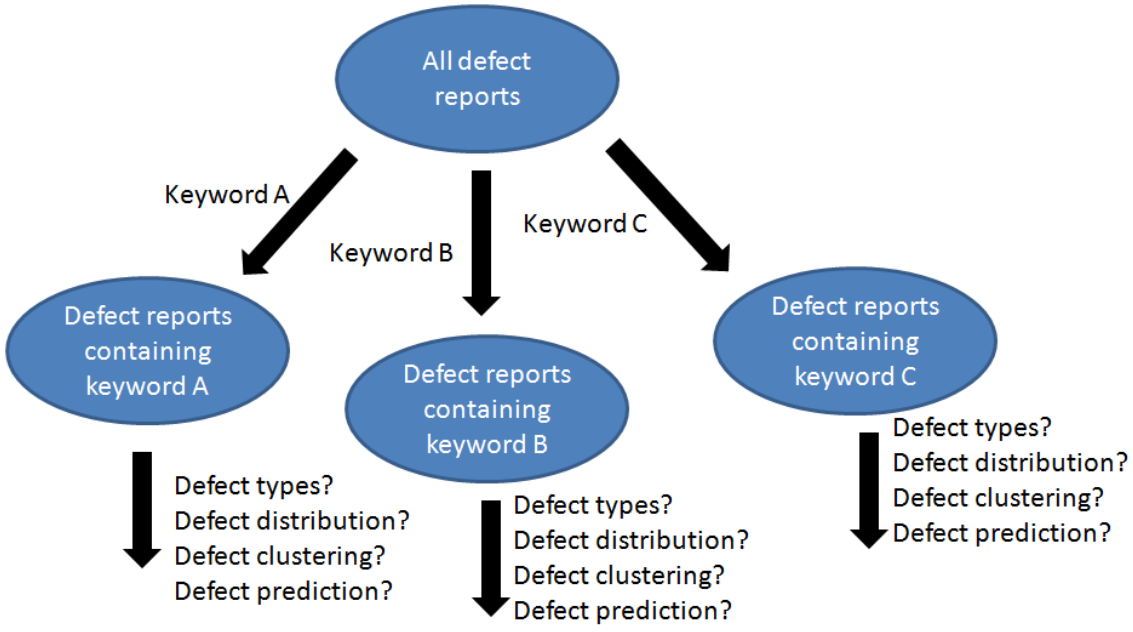


Figure 5-1: After defect classification?

Table 5.1: Fraction of defects by defect types for a sample project (total 423 defects)

Defect type (keyword)	Fraction of defects
range	2.36%
string	3.55%
javascript	4.49%
display	19.39%

Table 5.2: An example of defects indexed by the keyword **range**

Defects indexed by keywords	
Defect type (keyword)	Defect summary
range	Not able to link Contour project in JIRA's configuration ...
	Strange Login Information text for logging a Defect
	Invalid characters under Sub-Issues
	Date range chart not correct in history panel
	Filter dialog does not close in History panel
	Reviewers popup is not on top and has overlay problem with issue ...
	Strange layout of Review Findings list
	Link to a non-existing contour item cannot be deleted from JIRA
	Not able to approve "Review" for the "Review Finding Type" "Defect" ...
	User is unable to perform a self signup on Jira PROD

5.2 Distribution of defects in files

Once defects are indexed into groups, the distribution of defects in the associated files of each group becomes a question of interest. In particular, we are interested in the histogram of the fraction of files with defect counts. The histograms are computed by Algorithm 6 and Algorithm 7, and examples of histograms are shown in Figure 5-5. Discussion of experimental results is presented in Chapter 6. Compared with the previous study [28], our approach has additional benefits as we can study the distribution of defects for the defect types of interest just after indexing the defects by keywords.

5.3 Clustering of defects in files

By observing the distribution of defects in Figure 5-5, we see a few files have more defects than others. This gives the motivation to study the clustering of defects in files. In particular, we use two metrics, *ClusteringMetric* in Equation 5.2 and

Table 5.3: An example of defects indexed by the keywords **range**, **panel**, **contour**

Defects indexed by keywords	
Defect type (keywords)	Defect summary
range, **panel**	Items not displayed in History panel for Timeline view
	Date range chart not correct in history panel
range, **contour**	Not able to link Contour project in JIRA's configuration ...
	Link to a non-existing contour item cannot be deleted from JIRA

Algorithm 6 Defect distribution in files
Input: a list of files
Output: files sorted by number of defects

Require: x is a list of files
1: **function** DefectDistributionInSourceFiles(x)
2: $y \leftarrow$ SortFilesByDefectCountsInAscendingOrder(x)
3: $y \leftarrow$ ComputeFractionOfFilesWithDefectCounts(y)
4: **return** y

Fraction80Metric in Equation 5.3, to quantify the extent of defect clustering. *ClusteringMetric* was proposed by [28], and had the properties that distribution could be as random (*ClusteringMetric* = 1), less clustering than random (*ClusteringMetric* < 1), or more clustering than random (*ClusteringMetric* > 1). *Fraction80Metric* is adapted from the well known 20/80 rule in computer systems, 20% of the files have 80% of the defects. It computes the fraction of files containing 80% of the defects. The metrics are computed by Algorithm 8 and Algorithm 9, and examples of these metrics are shown in Table 5.4 and Table 5.5. Discussion of experimental results is presented in Chapter 6. Compared with the previous study [28], our approach has additional benefits as we can study the clustering of defects for the defect types of

Algorithm 7 Defect distribution in files for all defect types
Input: a list of files, a map of indexed defects by defect types
Output: files sorted by number of defects for all defect types

Require: x is a list of files, z is a map of the indexed defects
1: **function** DefectDistributionInSourceFilesForAllDefectTypes(x, z)
2: $y \leftarrow$ ClassifySourceFilesByDefectTypes(x, z)
3: **for** $y_i : y$ **do**
4: $z \leftarrow$ DefectDistributionInSourceFiles(y_i)
5: Output(z)

interest just after indexing the defects by keywords.

Algorithm 8 Defect clustering in files
Input: a list of files
Output: computed *ClusteringMetric*, *Fraction80Metric*

Require: x is a list of files
1: **function** DefectClusteringInSourceFiles(x)
2: $y \leftarrow$ DefectDistributionInSourceFiles(x)
3: $a \leftarrow$ ComputeClusteringMetric(y)
4: $b \leftarrow$ ComputeFraction80Metric(y)
5: **return** $\{a, b\}$

Algorithm 9 Defect clustering in files for all defect types
Input: a list of files, a map of classified defects by defect types
Output: computed *ClusteringMetric*, *Fraction80Metric* for all defect types

Require: x is a list of files, z is a map of the indexed defects
1: **function** DefectClusteringInSourceFilesForAllDefectTypes(x, z)
2: $y \leftarrow$ ClassifySourceFilesByDefectTypes(x, z)
3: **for** $y_i : y$ **do**
4: $\{a, b\} \leftarrow$ DefectClusteringInSourceFiles(y_i)
5: Output($\{a, b\}$)

$$ClusteringMetric = \frac{\sum_{i=1}^{n}(e_i - \mu)^2}{\sum_{i=1}^{n} e_i} \tag{5.2}$$

where

$\{e_1, e_2, ..., e_N\}$ denote the defects in N files

$\mu = \frac{\sum_{i=1}^{n} e_i}{N}$ is the average number of errors per file.

$$Fraction80Metric = \frac{\text{number of files containing 80\% of defects}}{\text{total number of files}} \tag{5.3}$$

where

number of files are counted from the files with the most defects.

Table 5.4: An example of *ClusteringMetric* by defect types

Defect type (keyword)	*ClusteringMetric*
All defects	17.44
reproduce	109.51
dialog	68.79
string	28.92

Table 5.5: An example of *Fraction80Metric* by defect types

Defect type (keyword)	Files	*Fraction80Metric*
All defects	1375	42.76%
reproduce	17	29.41%
dialog	51	41.18%
string	9	55.56%

5.4 Predicting the hot files of defects

After analyzing the clustering of defects in Table 5.4 and Table 5.5, we would like to find the hot files of defects where defects cluster. We used a known metric, *BugPredMetric* in Equation 5.4, to rank the files and evaluated its prediction accuracy by file cache simulation (Figure 5-6 [26]). *BugPredMetric* was proposed by [27]. The hot files are computed and evaluated by Algorithm 10 and Algorithm 11 for all defect types, and examples of these hot files are shown in Table 5.7 and Table 5.8. Note that the file cache replacement policy is illustrated in Algorithm 10, where the files are ranked by *BugPredMetric* and the top-ranked files are kept in the file cache. File cache simulation is evaluated by defect hit rate as shown in Equation 5.5. An example of file cache simulation results is in Table 5.6. Discussion of experimental results is

presented in Chapter 6. Compared with the previous studies [26, 27], our approach has additional benefits as we can compute the hot files for the defect types of interest just after indexing the defects by keywords.

Algorithm 10 Defect hot spots prediction in files
Input: a list of file commits, a file cache size, a slope parameter
Output: computed defect hit rate of the file cache replacement policy using a known bug prediction metric, a list of hot files in the file cache

Require: x is a list of file commits, c is the size of file cache, s is the slope parameter

```
1: function DefectHotSpotsPredictionInSourceFiles(x, c, s)
2:     hit ← 0
3:     miss ← 0
4:     defectHitRate ← 0.0
5:     cache ← New Set for storing files
6:     x ← SortCommitsInChronologicalOrder(x)
7:     for x_i : x do
8:         f ← ListOfCommitedFiles(x_i)
9:         for f_j : f do
10:            if cache contains f_j then
11:                hit ← hit + 1
12:            else
13:                miss ← miss + 1
14:            h ← UpdateCommitedFiles(f_j)
15:            r ← RankFilesByBugPredScore(h, s)   ▷ Compute BugPredMetric for
   each file, and sort the files in descending order
16:            cache ← UpdateFileCacheByScore(r, c) ▷ Keep the top c files in cache
17:    defectHitRate ← hit/(hit+miss)
18:    Output(cache)                                ▷ output the files in the file cache
19:    return hitRate
```

$$BugPredMetric = \sum_{i=1}^{N} \frac{1}{1 + e^{slope \cdot t_i}} \tag{5.4}$$

Algorithm 11 Defect hot spots prediction in files for all defect types
Input: a list of file commits, a file cache size, a slope parameter, a map of indexed defects by defect types
Output: computed defect hit rates for all defect types

Require: x is a list of file commits, c is the size of file cache, s is the slope parameter, z is a map of indexed defects by defect types

```
1: function DefectHotSpotsPredictionInSourceFilesForAllDefect-
   Types(x, c, s, z)
2:     y ← ClassifyCommitsByDefectTypes(x, z)
3:     for y_i : y do
4:         hitRate ← DefectHotSpotsPredictionInSourceFiles(y_i, c, s)
5:         Output(hitRate)
```

where

slope is a configuable parameter with default value 12.

N is the number of commits of a file.

t_i is the normalized timestamp of the ith commit,

with the oldest timestamp = 1 and now = 0.

$$\text{defect hit rate} = hit/(hit + miss) \tag{5.5}$$

where

hit is the number of times a commited file is in the file cache

miss is the number of times a commited file is not in the file cache

Table 5.6: An example of defect hit rate by defect types

file cache size = 20, and slope = 12		
Defect type (keyword)	Files	Defect hit rate
All defects	1375	30.66%
reproduce	17	100%
dialog	51	86.23%
string	9	100%

Table 5.7: Top-20 hot files for a sample project

atlassian-plugin.xml
ContourProjectManagerImpl.java
component-wise-issue-chart.js
changeHistory.js
ComponentWiseIssueReportResource.java
defect-resolution-time-chart.js
DefectResolutionTimeReportResource.java
chart.js
ComponentWiseIssueReport.java
ChangeHistoryTableImpl.java
ChangeHistoryChartResource.java
chart-utils.js
chart-common.js
ComponentWiseIssueChart.java
pom.xml
ContourCommonSoapManager.java
ContourProjectManager.java
chart-defaults.js
projectAndFilterPicker.js
defect-trend-chart.js

5.5 Summary

This chapter presented the methodology of integrating keyword indexing of defects with code commits by four techniques: inferring defect types by keywords, studying the distribution of defects in files, studying the clustering of defects in files, and predicting the hot spots of defects in files. The challenge of inferring defect types by keywords is to identify the proper keywords extracted from the defect reports, and the approximation power of keywords depends on the documentation quality of defect reports. Studying the distribution of defects in files by the histograms of the fraction

Table 5.8: Top-17 hot files indexed by the keyword **reproduce** for a sample project

atlassian-plugin.xml
ContourProjectManagerImpl.java
pom.xml
ContourCommonSoapManager.java
userpicker.js
ResolutionFieldsValidator.java
ReviewersAndApproversManagerImpl.java
BulkWatch.java
SetReviewersResult.java
SoftCoDeleteIssue.java
duplicateissuepicker_listener.js
contourprojectpicker.js
contourprojectpicker_listener.js
atlassian-plugin.xml
UpdateIssueCustomFieldPostFunction.java
SoftCoCascadingSelectCFType.java
SoftCoCascadingSelectCFType.java

of files with defect counts could give some insight into how defects are generated. Studying the clustering of defects in files shows that defects cluster in certain files. We quantify the extent of defect clustering by two metrics, *ClusteringMetric* and *Fraction80Metric*. Predicting the hot spots of defects is done by ranking the files with *BugPredMetric* and its accuracy is evaluated by file cache simulation. We can study the distribution & clustering, and predict the hot files of defects for the defect types of interest after indexing the defects by keywords.

Once the methodology is developed, we can perform studies on Honeywell ACS projects in details and give discussion and interpretation of the experimental results, in particular answering the research questions in Section 1.2. Chapter 6 evaluates the developed methodology on two Honeywell ACS projects.

```
<pattern>
  <name>Range, contour defects</name>
  <description>
    <match>
      <name>description</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>contour</keyword>
      </keywords>
    </match>
  </description>
  <summary>
    <match>
      <name>summary</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>contour</keyword>
      </keywords>
    </match>
  </summary>
  <comments>
    <match>
      <name>comment</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>contour</keyword>
      </keywords>
    </match>
  </comments>
</pattern>
```

Figure 5-2: A defect pattern with two keywords **range**, **contour**

```
<pattern>
  <name>Range, panel defects</name>
  <description>
    <match>
      <name>description</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>panel</keyword>
      </keywords>
    </match>
  </description>
  <summary>
    <match>
      <name>summary</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>panel</keyword>
      </keywords>
    </match>
  </summary>
  <comments>
    <match>
      <name>comment</name>
      <keywords>
        <keyword>range</keyword>
      </keywords>
      <keywords>
        <keyword>panel</keyword>
      </keywords>
    </match>
  </comments>
</pattern>
```

Figure 5-3: A defect pattern with two keywords **range**, **panel**

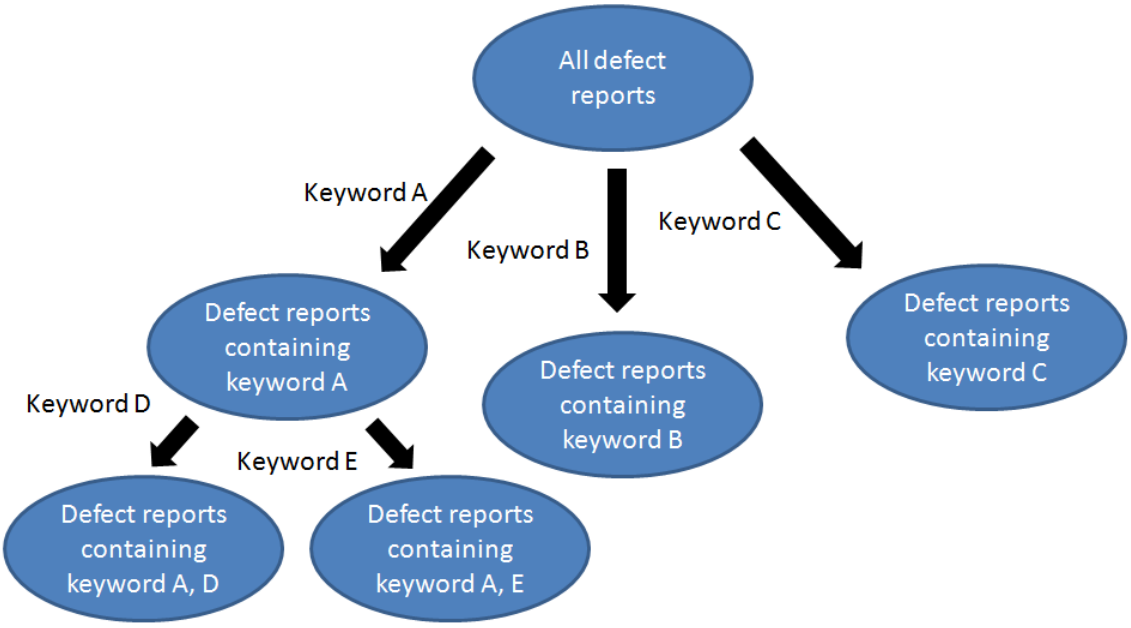


Figure 5-4: A defect classification tree

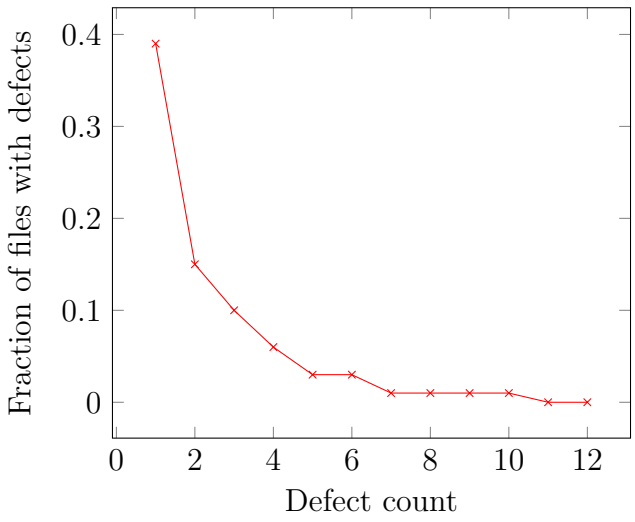


Figure 5-5: Histogram of the fraction of files with defect counts for a sample project (total 1909 files, 423 defects)

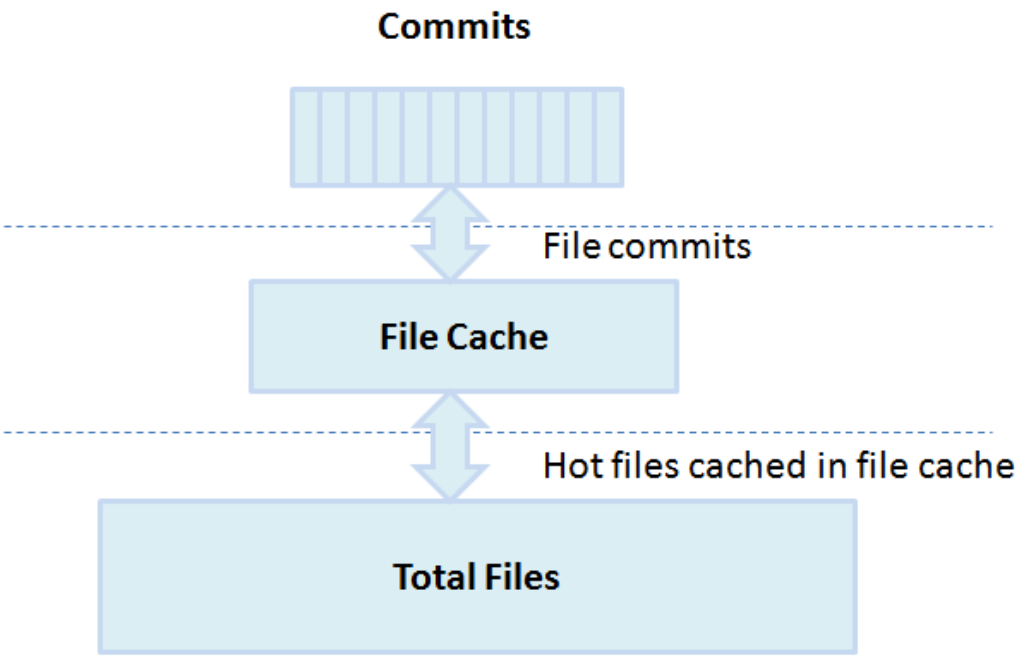


Figure 5-6: File cache simulation for evaluating hot files of defects

Chapter 6

Evaluation

> What you do in this world is a matter of no consequence. The question is what can you make people believe you have done.
>
> Arthur Conan Doyle, A Study in Scarlet

Chapter 5 presented the methodology of integrating defect indexing with code commits. This chapter evaluates the proposed methodology, and the knowledge behind it, on two Honeywell ACS projects. In each project, we infer the defect types by keywords, study the distribution & clustering of defects, and predict and the hot files of defects. Section 6.1 explains the evaluation strategy. Section 6.2 evaluates the proposed methodology on a JIRA server extension project. Section 6.3 evaluates the proposed methodology on a gas ignition controls project. Section 6.4 discusses the implications of the results. Finally, a summary in given in Section 6.5.

6.1 Evaluation strategy

Two Honeywell ACS projects with their descriptions and statistics are shown in Table 6.1. The projects were chosen by three criteria: 1) Programming Languages (PL), 2) defect data size, 3) applications. We expect projects using different programming

languages, having similar defect data size, and developed for different applications would have different defect characteristics. The two selected projects are: Project X, a JIRA server extension project developed in Java with 423 defects and 3235 commits, and Project Y, a gas ignition controls project developed in C with 300 defects and 2977 commits. We investigate the defect types, the defect distribution, the defect clustering, and the hot files of defects of these two projects in the following sections.

Table 6.1: Projects for evaluation

Project	Description	PL	Defects	Commits
Project X	a JIRA server extension	Java	423	3235
Project Y	Gas ignition controls	C	300	2977

6.2 Case study: A JIRA server extension project

What are the defect types?

We selected 10 keywords from the extracted keyword list of project X, and indexed the defects of project X by these keywords: **layout**, **string**, **javascript**, **reproduce**, **message**, **configuration**, **display**, **panel**, **dialog**, **report**. The results of indexing are shown in Table 6.2. Project X is a JIRA server extension project and these keywords could provide some insight into the root causes of the defects. The indexing power of these keywords is around 73%, with 27% defects unindexed. As the most dominant defect type, 40/64 defects indexed by the keyword **report** are shown in Table 6.3.

Distribution of defects in files

We observed the defect distribution of project X in Figure 6-1. Around 40% of the files contained 1 defect, 15% of the files contained 2 defects, 10% of the files contained

Table 6.2: Fraction of defects by defect types for project X (total 423 defects)

Defect type (keyword)	Fraction of defects
layout	1.42%
string	1.89%
javascript	1.89%
reproduce	4.49%
message	5.91%
configuration	6.15%
display	9.93%
panel	11.58%
dialog	13.71%
report	15.13%
Unindexed defects	27.19%

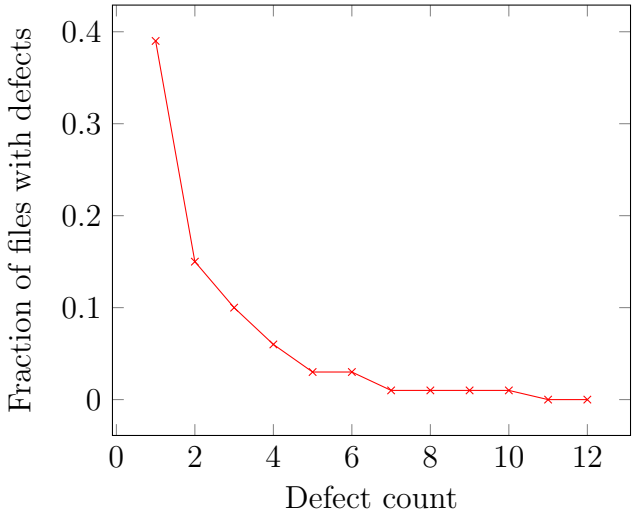


Figure 6-1: Histogram of the fraction of files with defect counts for project X (total 1909 files, 423 defects)

3 defects. Defects distributed non-uniformly and around 18% of the files contained no defects. The distribution of defects is on the same order of magnitude as the previous study [28].

Clustering of defects in files

We analyzed the clustering of defects in files of project X by computing *ClusteringMetric*. The results are shown in Table 6.4 6.5. Clustering of all defects is already strong (*ClusteringMetric* = 17.44 >> 1), and clustering by defect types is even stronger. Clustering of defects indexed by the keywords **javascript** (*ClusteringMetric*

= 140.07), **reproduce** (*ClusteringMetric* = 109.51), **configuration** (*ClusteringMetric* = 107.21) is particularly strong. Compared with the previous study [28], clustering is stronger as the computed *ClusteringMetric*-s are much larger.

We also verified the 20/80 rule for project X computing *Fraction80Metric*. The results are shown in Table 6.5. We observed for all defect types 20%-50% of files contained more than 80% of defects, with defects indexed by the keyword **javascript** followed the 20/80 rule.

Predicting the hot files of defects

We predicted the hot files of defects for project X by ranking the files by *BugPreMetric*. The top-20 hot files of all defects for project X are shown in Table 6.6, and the top-20 hot files for the defects indexed by the keyword **report** are shown in Table 6.7. Examples of these hot files are: ContourProjectManagerImpl.java, component-wise-issue-chart.js, and defect-resolution-time-chart.js.

We evaluated the accuracy of prediction by file cache simulation. Table 6.8 shows the the number of files by defect types. The results of simulation are shown in Table 6.9 using file cache size = 20 and slope = 12. Figure 6-2 shows the results of simulation using different file cache size and a fixed slope. We observed defect hit rate increased as file cache size increased, but enlarging file cache size also diluted the strength of the messages: A list of top-100 hot files might be too long to look into for some programmers.

The defect hit rate for all defects using file cache size = 20 is 30%, and it increases to 55% when file cache size = 100. The defect hit rates by defect types are much higher as the number of files are fewer: with most defect types having defect hit rates above 90%, and defects indexed by the keyword **dialog** around 86%. Compared with the previous study [26], the file cache performance is much better after sorting the files by defect types, supporting defect classification before defect prediction. The slope

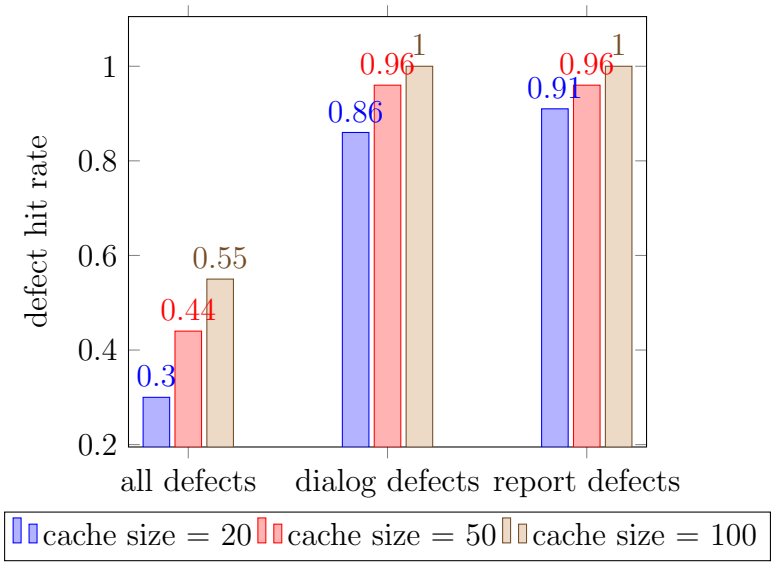


Figure 6-2: Defect hit rate with different file cache size and slope = 12 for project X

is fixed to 12 [27] in all experiments, and the optimization of the slope for better file cache performance is left for future work.

6.3 Case study: A gas ignition controls project

What are the defect types?

We selected 10 keywords from the extracted keyword list of project Y, and indexed the defects of project Y by these keywords: **print**, **string**, **memory**, **fan**, **cool**, **power**, **lock**, **circulator**, **alarm**, **heat**. The results of indexing are shown in Table 6.10. Project Y is a gas ignition controls project and these keywords could provide some insight into the root causes of the defects. The indexing power of these keywords is

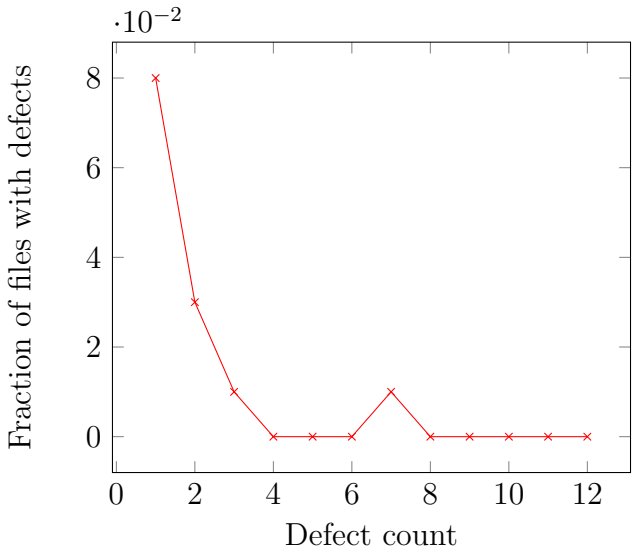


Figure 6-3: Histogram of the fraction of files with defect counts for project Y (total 18240 files, 300 defects)

around 62%, with 38% defects unindexed. As the most dominant defect type, 40/67 defects indexed by the keyword **heat** are shown in Table 6.11.

Distribution of defects in files

We observed the defect distribution of project Y in Figure 6-3. Around 8% of the files contained 1 defect, 3% of the files contained 2 defects, 1% of the files contained 3 defects. Defects distributed non-uniformly and around 85% of the files contained no defects. The fraction of files of the same defect counts is one magnitude less than the previous studies [28].

Clustering of defects in files

We analyzed the clustering of defects in files of project Y by computing *Clustering-Metric*. The results are shown in Table 6.12. The files for the defects indexed by the keyword **print** are not recorded so we omit this defect type. Clustering of all defects

is already strong (*ClusteringMetric* = 10.87 >> 1), and clustering by defect types is even stronger. Clustering of defects indexed by the keywords **string** (*ClusteringMetric* = 21.58), **alarm** (*ClusteringMetric* = 19.62), **lock** (*ClusteringMetric* = 18.91) is particularly strong. Compared with the previous study [28], clustering is stronger as the computed *ClusteringMetric*-s are much larger.

We also verified the 20/80 rule for project Y by computing *Fraction80Metric*. The results are shown in Table 6.13. The files for the defects indexed by the keyword **print** are not available and the files for the defects indexed by the keyword **memory** are too few, so we omit these two defect types. We observed for most defect types 20%-50% of files contained more than 80% of defects, with the defects indexed by the keyword **fan** having 73% of files contained 80% of defects, and the defects indexed by the keyword **cool** having 64% of files contained 80% of defects.

Predicting the hot files of defects

We predicted the hot files of defects for project Y by ranking the files by *BugPreMetric*. The top-20 hot files of all defects for project Y are shown in Table 6.14, and the top-20 hot files for the defects indexed by the keyword **heat** are shown in Table 6.15. Examples of these hot files are: application.c, ign_interface.c, and LED_UserInterface.c.

We evaluated the accuracy of prediction by file cache simulation. Table 6.16 shows the the number of files by defect types. The files for the defects indexed by the keyword **print** are not recorded so we omit this defect type. The results of simulation are shown in Table 6.17 using file cache size = 20 and slope = 12. Figure 6-4 shows the results of simulation using different file cache size and a fixed slope. We observed defect hit rate increased as file cache size increased, but enlarging file cache size also diluted the strength of the warning messages: A list of top-100 hot files might be too long to look into for some programmers.

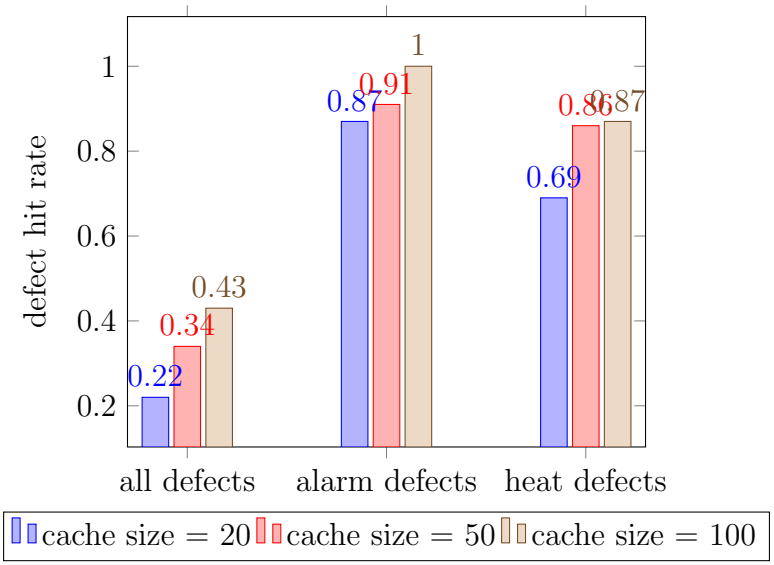


Figure 6-4: Defect hit rate with different file cache size and slope = 12 for project Y

The defect hit rate for all defects using file cache size = 20 is 22%, and it increases to 43% when file cache size = 100. The defect hit rates by defect types are much higher as the number of files are fewer: with most defect types having defect hit rates above 80%, and defects indexed by the keyword **heat** around 70%. Compared with the previous study [26], the file cache performance is much better after sorting the files by defect types, supporting defect classification before defect prediction. The slope is fixed to 12 [27] in all experiments, and the optimization of the slope for better file cache performance is left for future work.

6.4 Discussion

Projects X and Y are compared by their defect types in Table 6.18. Project X contains the defects related to **layout**, **string**, **javascript**, **reproduce**, **message**,

configuration, **display**, **panel**, **dialog**, **report**. Project Y contains the defects related to **print**, **string**, **memory**, **fan**, **cool**, **power**, **lock**, **circulator**, **alarm**, **heat**. These keywords could provide some insight into how defects were generated for projects X and Y. The defects not indexed by these keywords for projects X and Y are 27% and 38%, respectively.

Projects X and Y are compared by their defect distribution & clustering in Table 6.19. For project X, there are around 40% of the files contained 1 defect, 15% of the files contained 2 defects, 10% of the files contained 3 defects. Defects distributed non-uniformly and around 18% of the files contained no defects. Project X has a *ClusteringMetric* = 17.44, and for all of its defect types, 20%-50% of files contained more than 80% of defects. For project Y, there are around 8% of the files contained 1 defect, 3% of the files contained 2 defects, 1% of the files contained 3 defects. Defects distributed non-uniformly and around 85% of the files contained no defects. Project Y has a *ClusteringMetric* = 10.81, and for most of its defect types, 20%-50% of files contained more than 80% of defects. Overall, project X exhibits stronger defect clustering than project Y.

Projects X and Y are compared by their hot files of defects in Table 6.20. For project X, hot files are mostly Java and JavaScript source files, relating to project management, defect resolution, report and history functions. For project Y, hot files are C source or header files, relating to applications, scheduler functions, LED interface, data and errors.

Since projects X and Y are developed by different programming languages (Java vs. C) for different applications (server extension vs. controls), it makes no sense to compare which project is better (or worse). Nevertheless, our approach to studying defect characteristics proves to be effective for quite different projects. Compared with the previous study [28], our approach works well on a more comprehensive variety of defects found in more diversified projects.

6.5 Summary

This chapter evaluated the methodology of integrating defect indexing with code commits on two Honeywell ACS projects and showed that: 1) Using 10 keywords could index defects around 60%-70%. and the keywords could provide some insight into the root causes of the defects. 2) Defect distribution varied by projects, and could provide some insight into how defects were generated. 3) Defect clustering metrics could show the extent of clustering for the defects. 4) Hot files of defects by defect types had high prediction accuracy, supporting for defect classification before defect prediction. 5) Projects could be compared by defect distribution & clustering to understand the non-uniformity of defect distribution and the motivation of identifying the hot files of defects, and could be compared by defect types and hot files of defects to understand the roots causes of defects.

Table 6.3: 40/64 defects indexed by the keyword **report** for project X

Defects indexed by keywords	
Keyword	Defect summary
report	Exception in Defect Scorecard
	The Test Run is not related with a JIRA proxy object
	Exceptions related to Defect Profile and Defect Trend report
	Detected Vs Resolved Defect report is showing Negative values for Unresolved issues.
	Risk Register report throwing an expection when trying to generate the report
	Defect Profile Report is not providing any chart.Not able to find any logs in the server.
	Report Problem Link is still pointing to the old "Helpdesk" project.
	Defect Aging report is not showing any information and throwing exceptions in the logs.
	Defect Resolution Effectiveness report not showing any information in JIRA and ...
	Reported in Version field misses "Unknown" Value
	Defect scorecard for project "Monitor Upgrade" results in exception
	"Reported in Version" value does not populate "Reported in Version" field of Defect
	Field "Version" in create defect screen for test run should be labeled ...
	Exception in Defect Scorecard
	Error in favorite reports
	Defect Resolution Time Report gets stucked with IE
	Deferred defects not managed in Resolution Time Report
	Application gets stucked in Refine report
	Menu items covered by the query field
	Reviews with no findings are not displayed in the Reviews report
	Review column missing in Reviews report
	Total review number not correct in Review report
	Not able to create "Process Usage Report" if project is selected
	Contour item/s field content not properly shown in issue navigator column ...
	Unfinished features included in latest release
	Reporter field in "Create and Link" screen empty
	Icon Assign to Reporter in Issue Creation
	Defect Trend Report - graph legend shows incorrect labels
	Reports do not take into account findings of type Defect even after checking box ...
	JIRA User Signup throws exception upon invocation
	Bulk Editing of issues is not allowing to change the Assignee or Reporter
	Layout cosmetics in Detected vs Resolved report
	Link not clickable in chart of Detected vs Resolved defects report
	Refine report popup close when project is canceled
	Refine window does not open in Defect Trend report
	Resolution Effectiveness Report gets stucked
	Link to issue navigator doesn't work in effectiveness report
	Chart not shown in Detected vs Resolved report
	Defect numbers do not match in Resolution Effectiveness report
	Reporter field is empty on create issue screen (e.g., for task)

Table 6.4: *ClusteringMetric* by defect types for project X

Defect type (keyword)	*ClusteringMetric*
All defects	17.44
layout	96.86
string	28.92
javascript	140.07
reproduce	109.51
message	88.65
configuration	107.21
display	75.91
panel	77.89
dialog	68.79
report	75.28

Table 6.5: *Fraction80Metric* by defect types for project X

Defect type (keyword)	Number of files	*Fraction80Metric*
All defects	1375	42.76%
layout	17	35.29%
string	9	55.56%
javascript	15	20.00%
reproduce	17	29.41%
message	35	28.57%
configuration	16	31.25%
display	65	24.62%
panel	42	30.95%
dialog	51	41.18%
report	56	48.21%

Table 6.6: Top-20 hot files for project X

atlassian-plugin.xml
ContourProjectManagerImpl.java
component-wise-issue-chart.js
changeHistory.js
ComponentWiseIssueReportResource.java
defect-resolution-time-chart.js
DefectResolutionTimeReportResource.java
chart.js
ComponentWiseIssueReport.java
ChangeHistoryTableImpl.java
ChangeHistoryChartResource.java
chart-utils.js
chart-common.js
ComponentWiseIssueChart.java
pom.xml
ContourCommonSoapManager.java
ContourProjectManager.java
chart-defaults.js
projectAndFilterPicker.js
defect-trend-chart.js

Table 6.7: Top-20 hot files for the defects indexed by the keyword **report** for project X

atlassian-plugin.xml
ContourProjectManagerImpl.java
defect-resolution-time-chart.js
DefectResolutionTimeReportResource.java
chart.js
chart-utils.js
chart-common.js
projectAndFilterPicker.js
defect-trend-chart.js
userpicker.js
SoftCoConstants.java
DefectResolutionTimeReport.java
chart-filters.js
ProcessUsageReport.java
ContourSqlManagerImpl.java
DefectTrendReportResource.java
ProjectsAndFiltersResource.java
TimeUtils.java
SignupEx.java
DefectResolutionEffectivenessChart.java

Table 6.8: Number of files by defect types for project X

Defect type (keyword)	Number of files
All defects	1375
layout	17
string	9
javascript	15
reproduce	17
message	35
configuration	16
display	65
panel	42
dialog	51
report	56

Table 6.9: Defect hit rate by defect types for project X

file cache size = 20, and slope = 12		
Defect type (keyword)	Files	Defect hit rate
All defects	1375	30.66%
layout	17	100.00%
string	9	100.00%
javascript	15	100.00%
reproduce	17	100.00%
message	35	95.61%
configuration	16	100.00%
display	65	90.60%
panel	42	90.08%
dialog	51	86.23%
report	56	91.10%

Table 6.10: Fraction of defects by defect types for project Y (total 300 defects)

Defect type (keyword)	Fraction of defects
print	0.33%
string	1.00%
memory	1.67%
fan	2.67%
cool	3.00%
power	5.67%
lock	6.33%
circulator	7.00%
alarm	11.67%
heat	22.33%
Unindexed defects	38.33%

Table 6.11: 40/67 defects indexed by the keyword **heat** for project Y

Defects indexed by keywords	
Keyword	Defect summary
heat	Device report h and 55 on 90+model
	PS fault are not updated in fault history to level 2
	Device does not clear PS fault when is request removed when is in other fault state
	Device report h when is in High heating by RheemNet
	Firing rate in communicating defrost
	Gas valve 2nd stage relay can't make it definition problem
	Incorrect Heat Demand Handling When Engineering Mode Was Active
	After roll out open device turn on bad circulator
	Ater limit open problem in WPSC device go into prepurge without W request
	High pressure switch failed (stuck) Closed cleaning
	Device cycle low/Hi inducer after heat fan on delay when W1+W2 request ...
	Device set bad CFM for a moment after heat fan on delay at 90+model
	LPS open in TFI or runnig - Bad clearing
	Service demand with CF6 cleared
	False flame detected at 18VAC secondary voltage
	Time between two heat cycles
	HPS open in high fire rate
	Relay cycling when W1+W2 request at 90+ device
	Inducer speed and HPS proving when is fault 57 active
	Fault 57 level 2
	IFC reports h instead of H in communicating mode
	Blower speed drops when Limit opens
	Heat cool priorities in OEM test mode
	OEM test mode exiti conditions
	Soft lockout is entered due to HPS failure during several separate calls for heat
	No reaction to false flame when waiting for Open PS
	Error 131 is announced when non-communicating OEM test mode is entered
	CLONE -Error 131 is announced when non-communicating OEM test mode ...
	OEM Test Mode entering procedure is incorrectly implemented
	Blower CFM is not clamped properly
	Device ignore HPS during heat fan on delay
	Heating command is not clamped properly
	Heating is allowed with no Model Data after lockout reset
	Transition delays during OEM test mode are not bypassed
	CR6.630 & CR6.631: Behaviour after exit from OEM test mode ...
	Fault 58 reported in OEM test mode
	Heating mode during OEM test mode
	IFC shall light on high fire during OEM test mode
	d6 fault reported with Fault level 2
	Error code 7+5 during EFT testing

Table 6.12: *ClusteringMetric* by defect types for project Y

Defect type (keyword)	*ClusteringMetric*
All defects	10.87
string	21.58
memory	7.71
fan	13.27
cool	18.48
power	8.28
lock	18.91
circulator	17.25
alarm	19.62
heat	15.32

Table 6.13: *Fraction80Metric* by defect types for project Y

Defect type (keyword)	Number of files	*Fraction80Metric*
All defects	2736	7.27%
string	25	56.00%
fan	30	73.33%
cool	50	64.00%
power	18	33.33%
lock	53	52.83%
circulator	76	44.74%
alarm	64	34.38%
heat	104	55.77%

Table 6.14: Top-20 hot files for project Y

application.c
ign_interface.c
osdata.h
IgnSupport.c
RN_ObjectProc.c
LED_UserInterface.c
trunk/LED_UserInterface.c
error.c
appscheduler.c
ModelDataCheck.c
RN_ObjectProc.c
RN_FaultsSetup.h
FanComm.h
Mod_IFC_2-STAGE_DBG_ignTab.h
branch/application.c
Mod_IFC_MODULATING_DBG_ignTab.h
ecmCOM.c
branch/ign_interface.c
trunk/ign_interface.h
Valves.c

Table 6.15: Top-20 hot files for the defects indexed by the keyword **heat** for project Y

application.c
ign_interface.c
osdata.h
LED_UserInterface.c
error.c
ModelDataCheck.c
RN_FaultsSetup.h
Mod_IFC_2-STAGE_DBG_ignTab.h
Mod_IFC_MODULATING_DBG_ignTab.h
branch/ign_interface.c
ign_interface.h
ecmCOM.c
Mod_IFC_2-STAGE_ignTab.h
StatAlgMod.c
branch2/ign_interface.c
OEMTest.c
slottasks.c
Mod_IFC_MODULATING_ignTab.h
RN_Faults.c
ECMFan.c

Table 6.16: Number of files by defect types for project Y

Defect type (keyword)	Number of files
All defects	2736
string	25
memory	3
fan	30
cool	50
power	18
lock	53
circulator	76
alarm	64
heat	104

Table 6.17: Defect hit rate by defect types for project Y

file cache size = 20, and slope = 12		
Defect type (keyword)	Files	Defect hit rate
All defects	1375	22.37%
string	25	88.25%
memory	3	100.00%
fan	30	86.59%
cool	50	90.04%
power	18	100.00%
lock	53	85.79%
circulator	76	81.32%
alarm	64	87.78%
heat	104	69.71%

Table 6.18: Comparison of defect types for projects X and Y

Project	Defect types (keywords)
Project X	**layout**, **string**, **javascript**, **reproduce**, **message**, **configuration**, **display**, **panel**, **dialog**, **report**, 27% defects unindexed.
Project Y	**print**, **string**, **memory**, **fan**, **cool**, **power**, **lock**, **circulator**, **alarm**, **heat**, 38% defects unindexed.

Table 6.19: Comparison of defect distribution & clustering for projects X and Y

Project	Defect distribution	Defect clustering
Project X	Around 40% of the files contained 1 defect, 15% of the files contained 2 defects, 10% of the files contained 3 defects. Defects distributed non-uniformly and around 18% of the files contained no defects.	$ClusteringMetric = 17.44$, and for all defect types 20%-50% of files contained more than 80% of defects.
Project Y	Around 8% of the files contained 1 defect, 3% of the files contained 2 defects, 1% of the files contained 3 defects. Defects distributed non-uniformly and around 85% of the files contained no defects.	$ClusteringMetric = 10.81$, and for most defect types 20%-50% of files contained more than 80% of defects.

Table 6.20: Comparison of hot files of defects for projects X and Y

Project	Hot files of defects
Project X	atlassian-plugin.xml
	ContourProjectManagerImpl.java
	component-wise-issue-chart.js
	changeHistory.js
	ComponentWiseIssueReportResource.java
	defect-resolution-time-chart.js
	DefectResolutionTimeReportResource.java
	chart.js
	ComponentWiseIssueReport.java
	ChangeHistoryTableImpl.java
Project Y	application.c
	ign_interface.c
	osdata.h
	IgnSupport.c
	RN_ObjectProc.c
	LED_UserInterface.c
	trunk/LED_UserInterface.c
	error.c
	appscheduler.c
	ModelDataCheck.c

Chapter 7

Implementation

The devil is in the detail.

Anonymous

7.1 Java implementation

We implemented the defect data integration tool, bugc, using the Java programming language. An open source XML parser software, MWDumper [43], is integrated in bugc as the frontend parser for defect reports and code commits in XML format.

The main program operation is illustrated by the excerpt of the main Java program below. For more details, the interested readers are invited to explore the source code provided in the release package.

```
try {
  // Read defect reports
  if (issueInputList.size() != 0) {
    XmlDumpReader2 reader = new XmlDumpReader2(issueInputList,
                                    bugAnalyzer);
    reader.readDump();
  }
  else {
```

```
  XmlDumpReader2 reader = new XmlDumpReader2(input, bugAnalyzer);
  reader.readDump();
}
// Read defect pattern
if (bugpattern != null) {
  BugWriter bugPatCollector = new BugPatternWriter();
  XmlBugPatternReader bugReader = new XmlBugPatternReader(bugpattern,
                                        bugPatCollector);
  bugReader.readDump();
  List<BugPattern> bpList = bugPatCollector.readBugPattern();
  bugPatCollector.close();
  bugAnalyzer.writeBugPattern(bpList);
}
// Read code commits data
if (commitInputList.size() != 0) {
  XmlCommitsWriter commitCollector = new HoneywellACSCommitsWriter();
  XmlCommitsReader commitReader = new XmlCommitsReader(commitInputList,
                                        commitCollector);
  commitReader.readDump();
  List<FisheyeCommitItem> commitList = commitCollector.readCommit();
  commitCollector.close();
  bugAnalyzer.writeCommitItem(commitList);
}
// Read code reviews data
if (reviewInputList.size() != 0) {
  XmlReviewsWriter reviewCollector = new HoneywellACSReviewsWriter();
  XmlReviewsReader reviewReader = new XmlReviewsReader(reviewInputList,
                                        reviewCollector);
  reviewReader.readDump();
  List<CrucibleReviewItem> reviewList = reviewCollector.readReview();
```

```
    reviewCollector.close();
    bugAnalyzer.writeReviewItem(reviewList);
  }
  // Defect indexing
  if (!indexMode.equals("")) {
    bugAnalyzer.writeIndexCtrl(indexMode, indexDir);
  }
  // File cache simulation
  if (simCacheSize != 0 && simSlope != 0) {
    bugAnalyzer.writeSimBugCacheCtrl(simCacheSize, simSlope);
  }
  // Defect analysis and prediction
  bugAnalyzer.run();
  bugAnalyzer.close();
} catch (RuntimeException e) {
  System.out.println(e.getMessage());
}
```

7.2 Examples

Examples on how to use bugc are provided in the main script, run.bat. The following is an excerpt of the script. The interested readers are invited to explore the scripts provided in the release package.

```
:: project X


:: Keyword extraction
jara -jar bugc.jar --format=defect:X_train.out
  --bugpattern=.\pat\bugpattern_X.xml
  --issues .\input\issues\X --index=train:.\ml\train
java -jar bugc.jar --format=defect:X_test.out
```

```
  --bugpattern=.\pat\bugpattern_X2.xml
  --issues .\input\issues\X --index=test:.\ml\test
cd .\kea-5.0_full
runTestKea2
cd ..
java -jar bugc.jar --format=defect:X_collect.out
  --bugpattern=.\pat\bugpattern_X2.xml
  --issues .\input\issues\X --index=collect:.\ml\test

:: Defect analysis and prediction
java -jar bugc.jar --format=defect:X.out
  --bugpattern=.\pat\bugpattern_X.xml
  --issues .\input\issues\X --commits .\input\commits\X
:: file cache simulation (file cache size = 50, slope = 12)
java -jar bugc.jar --format=defect:X_cache_50_12.out
  --bugpattern=.\pat\bugpattern_X.xml
  --issues .\input\issues\X
  --commits .\input\commits\X --simcache=50:12
```

7.3 Command line arguments

Command line arguments of bugc are shown in Table 7.1. For more examples of these command line arguments, the interested readers are invited to explore the scripts provided in the release package.

7.4 Summary

This chapter introduced the implementation of a defect data integration tool, bugc, developed as a substrate for the experiments in this thesis. Many implementation

Table 7.1: Command line arguments

Arguments	Description	Default value
–format=defect:output-filename	specify the output file	none
–bugpattern=defect-pattern-filename	specify the defect pattern	none
–issues issue-directory	specify the directory of defect reports	none
–commits commits-directory	specify the directory of code commits	none
–index=index-mode:output-directory	specify indexing control parameters	none
–simcache=cache-size:slope	specify file cache simulation parameters	cache-size = 50, slope = 12

details are not covered in this chapter, and the interested readers are invited to explore the source code and scripts provided in the release package.

Chapter 8

Conclusions and future work

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> Donald E. Knuth

8.1 Conclusions

This thesis examines the integration of the three-dimensional defect data found in defect reports, code reviews and code commits. Due to lacking of enough code reviews data, this thesis focuses on the links between defect reports and code commits by revolving around five central research questions (Section 1.2). By identifying defect classification as the key to answering the research questions, this thesis proposes the methodology of indexing defects by keywords and the methodology of studying the properties of the indexed defects. The methodology is evaluated on two Honeywell ACS projects: projects X and Y.

What are the defect types?

Defect types of the selected sample projects are inferred by keywords. Project X contains the defects related to **layout** (1.42%), **string** (1.89%), **javascript** (1.89%), **reproduce** (4.49%), **message** (5.91%), **configuration** (6.15%), **display** (9.93%),

panel (11.58%), **dialog** (13.71%), **report** (15.13%). Project Y contains the defects related to **print** (0.33%), **string** (1.00%), **memory** (1.67%), **fan** (2.67%), **cool** (3.00%), **power** (5.67%), **lock** (6.33%), **circulator** (7.00%), **alarm** (11.67%), **heat** (22.33%). These keywords were extracted by our automatic tools that were far more efficient than manual reviews, and could provide some insight into how the defects were generated for projects X and Y. The defects not indexed by these keywords for projects X and Y are 27% and 38%, respectively.

How are defects distributed in files?

Defect distribution is computed after indexing the defects by keywords. For project X, there are around 40% of the files contained 1 defect, 15% of the files contained 2 defects, 10% of the files contained 3 defects. The defects distribute non-uniformly and around 18% of the files contain no defects. For project Y, there are around 8% of the files contained 1 defect, 3% of the files contained 2 defects, 1% of the files contained 3 defects. The defects distribute non-uniformly and around 85% of the files contain no defects.

Do defects cluster in files?

Defect clustering metrics are computed after indexing the defects by keywords. The answer to this question is YES. Project X has a *ClusteringMetric* = 17.44, and for all of its defect types, 20%-50% of files contained more than 80% of defects. Project Y has a *ClusteringMetric* = 10.81, and for most of its defect types, 20%-50% of files contained more than 80% of defects. Overall, project X exhibits stronger defect clustering than project Y.

How to predict the hot files of defects?

Hot files of defects are predicted, by ranking the files by *BugPredMetric*, after indexing the defects by keywords. For project X, the hot files are mostly Java and JavaScript source files, relating to project management, defect resolution, report and

history functions. For project Y, the hot files are C source or header files, relating to applications, scheduler functions, LED interface, data and errors.

How do different projects compare in terms of defect characteristics?

The selected projects are compared after analyzing the types, distribution, clustering and hot files of the defects for them. Since projects X and Y are developed by different programming languages (Java vs. C) for different applications (server extension vs. controls), it makes no sense to compare which project is better (or worse). Nevertheless, our approach to studying defect characteristics proves to be effective for quite different projects. Compared with the previous study [28], our approach works well on a more comprehensive variety of defects found in more diversified projects.

8.2 Future work

This thesis answered the five research questions addressing the links between defect reports and code commits. However, there are still many interesting questions left unexplored. Some of them are suggested as follows.

Towards dynamic defect analysis

Indexing defects by keywords provides a means to classify defects effectively. It would be interesting to use this technique to study the time evolution of defects in different stages of a project. 1) Would the dominant defect types vary through time? 2) Would defect distribution, defect clustering, and hot files of defects vary through time? 3) What would be the implications of defect evolution through time?

Integrating defect data online: JIRA plugin

The developed defect data integration tool, bugc, operates as a command line tool. This puts some restrictions on the deployment of this tool on a global scale. It

would be worthwhile to develop an online version of bugc as a plugin to a commercial defect tracking system (JIRA [2]). This tool can be directly deployed on a central server, and used by users at Honeywell ACS sites around the world. For this online defect data integration tool, some interesting questions are: 1) How to incorporate professional knowledge of defects from users into the keyword extraction process? 2) How to get feedback from users and optimize the performance of the tool? 3) How to encourage the collection of more complete defect data (defect reports, code commits, code reviews) to expand the coverage of the tool?

Bibliography

[1] Olena Medelyan, *Human-competitive automatic topic indexing*, Ph.D. thesis, University of Waikato, 2009.

[2] Atlassian, "Plan, track, work smarter and faster," `http://www.atlassian.com/software/jira`.

[3] Atlassian, "Collaborative peer code review," `http://www.atlassian.com/software/crucible`.

[4] Atlassian, "Search, track, and visualize code changes," `http://www.atlassian.com/software/fisheye`.

[5] Eibe Frank, Gordon W. Paynter, Ian H. Witten, Carl Gutwin, and et al., "Domain-specific keyphrase extraction," in *PROC. SIXTEENTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*. 1999, pp. 668–673, Morgan Kaufmann Publishers.

[6] Digital Libraries and Machine Learning Labs at Computer Science Department at The University of Waikato, "Kea: Keyword extraction algorithm," `http://www.nzdl.org/Kea`.

[7] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *Software Engineering, IEEE Transactions on*, vol. 18, no. 11, pp. 943–956, 1992.

[8] Norman E. Fenton and Niclas Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[9] Alexander Aiken, Manuel Fähndrich, and Zhendong Su, "Detecting races in relay ladder logic programs," 1998.

[10] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *In Network and Distributed System Security Symposium*, 2000, pp. 3–17.

[11] J. Gray, "A census of tandem system availability between 1985 and 1990," *Reliability, IEEE Transactions on*, vol. 39, no. 4, pp. 409–418, 1990.

[12] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992, pp. 475–484.

[13] Inhwan Lee and R.K. Iyer, "Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, 1993, pp. 20–29.

[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001.

[15] Yichen Xie and Dawson Engler, "Using redundancies to find errors," in *IEEE Transactions on Software Engineering*, 2002, pp. 51–60.

[16] Benjamin Chelf, Dawson Engler, and Seth Hallem, "How to write system-specific, static checkers in metal," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 1, pp. 51–60, Nov. 2002.

[17] Yichen Xie, Andy Chou, and Dawson Engler, "Archer: using symbolic, path-sensitive analysis to detect memory access errors," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 327–336, Sept. 2003.

[18] Dawson Engler and Ken Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003.

[19] Dawson Engler and Madanlal Musuvathi, "Static analysis versus software model checking for bug finding," in *In VMCAI*. 2004, pp. 191–210, Springer.

[20] Ted Kremenek and Dawson Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *In Proceedings of 10th Annual International Static Analysis Symposium*. 2003, pp. 295–315, Springer.

[21] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler, "Correlation exploitation in error ranking," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, New York, NY, USA, 2004, SIGSOFT '04/FSE-12, pp. 83–93, ACM.

[22] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson Engler, "From uncertainty to belief: Inferring the specification within," in *In "Proceedings of the Seventh Symposium on Operating Systems Design and Implemetation.* 2006, pp. 161–176, USENIX Association.

[23] Chris F. Kemerer and Mark C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, July 2009.

[24] Mika V. Mantyla and Casper Lassenius, "What types of defects are really discovered in code reviews?," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 430–448, May 2009.

[25] Nachiappan Nagappan and Thomas Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference*

on Software engineering, New York, NY, USA, 2005, ICSE '05, pp. 284–292, ACM.

[26] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu, "Bugcache for inspections: hit or miss?," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, New York, NY, USA, 2011, ESEC/FSE '11, pp. 322–331, ACM.

[27] C. Lewis and R. Ou, "Bug prediction at google," `http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html`, 2011.

[28] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001, SOSP '01, pp. 73–88, ACM.

[29] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, "Cp-miner: a tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, OSDI'04, pp. 20–20, USENIX Association.

[30] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou, "/*icomment: bugs or bad comments?*/," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, New York, NY, USA, 2007, SOSP '07, pp. 145–158, ACM.

[31] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram, "How do fixes become bugs?," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, New York, NY, USA, 2011, ESEC/FSE '11, pp. 26–36, ACM.

[32] Michael Gumowski, "Improving quality of collected defect data and defect classification process," M.S. thesis, University of Geneva, 2012.

[33] Peter D. Turney, "Learning to extract keyphrases from text," *CoRR*, vol. cs.LG/0212013, 2002.

[34] Ian H. Witten, Gordon W. Paynter, Eibe Frank, Carl Gutwin, and Craig G. Nevill-Manning, "Kea: Practical automatic keyphrase extraction," in *IN PROCEEDINGS OF THE 4TH ACM CONFERENCE ON DIGITAL LIBRARIES*, 1998, pp. 254–255.

[35] Ken Barker and Nadia Cornacchia, "Using noun phrase heads to extract document keyphrases," in *Proceedings of the 13th Biennial Conference of the Canadian Society on Computational Studies of Intelligence: Advances in Artificial Intelligence*, London, UK, UK, 2000, AI '00, pp. 40–52, Springer-Verlag.

[36] A. Hulth, *Combining machine learning and natural language processing for automatic keyword extraction*, Ph.D. thesis, Stockholm University, 2004.

[37] Julie Beth Lovins, "Development of a stemming algorithm," *Mechanical translation and computational linguistics*, 1968.

[38] Pedro Domingos and Michael Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Mach. Learn.*, vol. 29, no. 2-3, pp. 103–130, Nov. 1997.

[39] Machine Learning Group at Computer Science Department at The University of Waikato, "Weka 3: Data mining software in java," `http://www.cs.waikato.ac.nz/ml/weka`.

[40] Peter D. Turney, "Coherent keyphrase extraction via web mining," in *Proceedings of the 18th international joint conference on Artificial intelligence*, San Francisco, CA, USA, 2003, IJCAI'03, pp. 434–439, Morgan Kaufmann Publishers Inc.

[41] Thuy Dung Nguyen and Min-Yen Kan, "Keyphrase extraction in scientific publications," in *Proceedings of the 10th international conference on Asian digital libraries: looking back 10 years and forging new frontiers*, Berlin, Heidelberg, 2007, ICADL'07, pp. 317–326, Springer-Verlag.

[42] Andras Csomai and Rada Mihalcea, "Linguistically motivated features for enhanced back-of-the-book indexing," in *ACL 2008, Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics, June 15-20, 2008, Columbus, Ohio, USA*. 2008, pp. 932–940, The Association for Computer Linguistics.

[43] MediaWiki developers, "Mwdumper: a quick little tool for extracting sets of pages from a mediawiki dump file," `http://www.mediawiki.org/wiki/Mwdumper`.