JDF: Detecting Duplicate Bug Reports in Jazz[*]

Yoonki Song[1] Xiaoyin Wang[2,3] Tao Xie[1] Lu Zhang[2,3] Hong Mei[2,3]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, USA
[2]Institute of Software, School of Electronics Engineering and Computer Science
[3]Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{ysong2, txie}@ncsu.edu, {wangxy06, zhanglu, meih}@sei.pku.edu.cn

ABSTRACT

Both developers and users submit bug reports to a bug repository. These reports can help reveal defects and improve software quality. As the number of bug reports in a bug repository increases, the number of the potential duplicate bug reports increases. Detecting duplicate bug reports helps reduce development efforts in fixing defects. However, it is challenging to manually detect all potential duplicates because of the large number of existing bug reports. This paper presents JDF (representing Jazz Duplicate Finder), a tool that helps users to find potential duplicates of bug reports on Jazz, which is a team collaboration platform for software development and process management. JDF finds potential duplicates for a given bug report using natural language and execution information.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement

General Terms: Management, Reliability

Keywords: bug report, execution information, information retrieval

1. INTRODUCTION

Bug reports play an important role in software development and maintenance. A common activity when using a bug repository is to detect duplicate bug reports stored in the bug repository. Detecting duplicate bug reports helps reduce the efforts of programmers in fixing related defects. However, it is quite challenging to manually detect duplicate bug reports since there can be a large number of bug reports.

Jazz [2] is a team collaboration platform for software development and process management. Jazz provides a repository for archiving software artifacts such as change sets, build results, and work items used in a development process. A work item represents a traceable and coordinated team work such as *Defect* (bug report). A bug report in a Jazz repository contains the same properties (such as summary and description) as those of common bug reports such as Bugzilla reports [1].

[*]This work is supported in part by an IBM Jazz Innovation Award, NSF grants CCF-0725190 and CCF-0845272, and ARO grants W911NF-08-1-0443 and W911NF-08-1-0105.

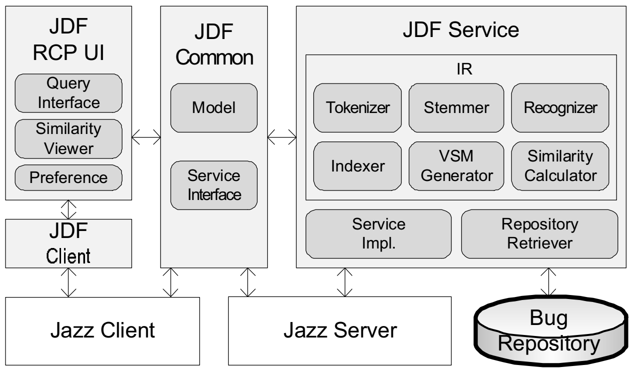


Figure 1: Architecture of the JDF tool

When submitting bug reports, Jazz users need to find potential duplicates to avoid submitting duplicates. To address this issue, we developed JDF[1] (representing Jazz Duplicate Finder), a tool that automatically mines a Jazz bug repository under analysis and detects duplicate bug reports. Our JDF tool uses natural language and execution information associated with the bug reports [7].

Rational Team Concert (RTC) [5], a Jazz extension that provides a collaborative software development platform, enables users to find duplicates of bug reports. RTC uses the summary and the description of a bug report in finding duplicates. Similar to Runeson et al.'s work [6], RTC gives a higher weight on the summary than that on the description of a bug report, because the summary is likely to contain more relevant words. Our JDF tool used several heuristics with different parameters including the parameter used in Runeson et al.'s work. In addition, RTC calculates similarities by using natural language information only, while our JDF tool computes similarities by using both natural language information and execution information.

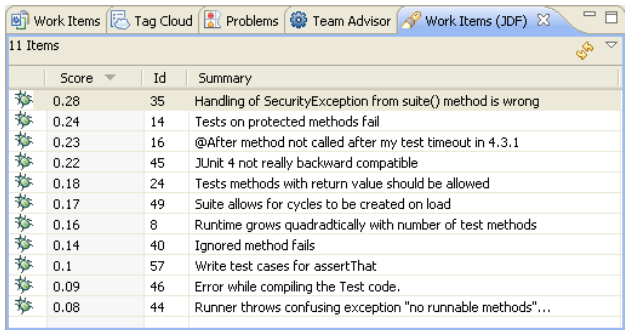
2. JDF ARCHITECTURE

Figure 1 provides the high-level overview of JDF's architecture. The architecture consists of four plug-ins: the *JDF Rich Client Platform (RCP) UI*, *JDF Client*, *JDF Common*, and *JDF Service*. JDF takes a bug report as input and produces a list of bug reports that are similar to the given bug report. We next present how the JDF client component and the JDF server component work and interact with each other.

2.1 JDF Client Component

The JDF client component contains the *JDF RCP UI*, *JDF Client*, and *JDF Common*. *JDF RCP UI* provides the Query Interface, the

[1]JDF is available at `http://sites.google.com/site/asergrp/projects/jazz`


Work Items | Tag Cloud | Problems | Team Advisor | Work Items (JDF)

11 Items

Score	Id	Summary
0.28	35	Handling of SecurityException from suite() method is wrong
0.24	14	Tests on protected methods fail
0.23	16	@After method not called after my test timeout in 4.3.1
0.22	45	JUnit 4 not really backward compatible
0.18	24	Tests methods with return value should be allowed
0.17	49	Suite allows for cycles to be created on load
0.16	8	Runtime grows quadradtically with number of test methods
0.14	40	Ignored method fails
0.1	57	Write test cases for assertThat
0.09	46	Error while compiling the Test code.
0.08	44	Runner throws confusing exception "no runnable methods"...

Figure 2: JDF View showing the result

Similarity Viewer as shown in Figure 2, and the Preference. The Preference provides a configuration interface for users to choose indexing schemes, similarity metrics such as Cosine and Jaccard [3], and which properties of a bug report should be considered in finding duplicates. JDF allows users to give weights on the summary and description of bug reports. For example, a user can find potential duplicates with the summary and the description of bug reports using the Cosine similarity metric and the same weight on the summary and the description of bug reports. *JDF Client* provides a mechanism for finding the service provided by *JDF Service* (described in Section 2.2) running on the Jazz Server. *JDF Common* provides models and service interfaces that are used by both the JDF client and JDF server sides.

Workflow among Components. If there is a JDF service request from the *JDF RCP UI*, the *JDF Client* finds the service through the Service Interface in the *JDF Common*. Then, the *JDF Common* calls the actual service in the *JDF Service*. After finding potential duplicates, the *JDF Service* returns the result to the *JDF RCP UI* through the *JDF Common*. Finally, the *JDF RCP UI* displays the result in the *Similarity Viewer*.

2.2 JDF Server Component

The JDF server component contains the *JDF Common* and *JDF Service*. As described in Section 2.1, the *JDF Common* is a common part of both the JDF client component and the JDF server component. The *JDF Service* contains an Information Retrieval (IR) implementation that is the main part of JDF's underlying approach [7], the Service Implementation for sending the result, and the Repository Retriever for retrieving existing bug reports from the repository. When retrieving existing bug reports, the Repository Retriever gathers only work items that are bug reports and in the same project area as the given bug report.

The Storage of Execution Information. The execution information of bug reports (in the form of method-call lists) is often very large (megabytes for one bug report). Therefore, to store execution information of all bug reports, we record a dynamic method list for each project. The elements in the dynamic method list are the methods (in the project) that have been executed at least once in the bug-revealing runs of all the existing bug reports. Thus, execution information of a bug report can be represented as a list of boolean values (called the execution list). Each boolean value in the execution list indicates whether the bug report involves calls to the corresponding method in the dynamic method list. When a new method is executed in a bug-revealing run of a newly submitted bug report, we add this new method to the end of the dynamic method list, and add a boolean value '0' to the end of the execution list of each existing bug report.

The IR Implementation. The IR implementation consists of six sub-components: the Tokenizer, Stemmer, Recognizer, Indexer, Vector Space Model (VSM [4]) Generator, and Similarity Calculator. More specifically, given the extracted text from the summary and/or the description of a bug report, the Tokenizer splits the text into tokens, and then the Stemmer identifies the root form from each token. Next, the Recognizer removes stop words such as "the" and "a" from the root forms. With the resulting root forms, the Indexer and the VSM Generator represent the vector space model. Finally, the Similarity Calculator produces the similarity scores between a given bug report and other bug reports in the repository using similarity metrics.

Main Idea for Calculating Similarity. Our technique [7] for the similarity calculator component consists of three steps. First, we calculate the Natural-Language-based Similarities (NL-S) between the new bug report and existing bug reports. Second, we calculate the Execution-information based Similarities (E-S) between the new bug report and existing bug reports. Finally, we retrieve potential target reports using the preceding two kinds of similarities based on two heuristics. The first heuristic is to combine the NL-S and the E-S into one combined similarity, and use the combined similarity to retrieve potential target reports. The second heuristic is to try to distinguish whether the natural language information or the execution information is the dominant factor in detecting each pair of possible duplicate reports, and use different strategies to deal with different situations. We next present our heuristics for combining the calculated similarities to retrieve potential duplicates of a bug report.

Retrieving Potential Duplicates. After calculating the NL-S score and the E-S score, we apply two heuristics (i.e., Basic Heuristic and Classification-Based Heuristic [7]) for ranking the existing bug reports in a list using these similarity scores. The Similarity Calculator uses the basic heuristic that gets the arithmetic average of the NL-S and E-S scores. When one of the two similarity scores is dominant, the calculator can use the classification-based heuristic with Credibility Threshold (CT) [7].

3. CONCLUSION

We have presented JDF, a set of Eclipse plug-ins integrated within Jazz to help users find potential duplicates using natural language and execution information. JDF also allows users to choose various indexing schemes and similarity metrics. JDF consists of the client component, which provides the user interface and the result view; and the server component, which retrieves a list of bug reports in a Jazz repository and computes similarity scores.

4. REFERENCES

[1] K. Herzig and A. Zeller. Mining the Jazz Repository: Challenges and Opportunities. In *Proc. MSR*, pages 159–162, 2009.

[2] Jazz. `http://jazz.net`.

[3] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[4] V. Raghavan and M. Wong. A Critical Analysis of Vector Space Model for Information Retrieval. *JASIS*, 37(5): 279–287, 1986.

[5] IBM Rational Team Concert. `http://www.ibm.com/software/awdtools/rtc/`.

[6] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proc. ICSE*, pages 499–510, 2007.

[7] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information. In *Proc. ICSE*, pages 461–470, 2008.