2014

Aarhat Multidisciplinary International Education Research Journal (AMIERJ)

(Bi-Monthly)
Peer-Reviewed Journal
Impact factor: 0.948

VOL - III Issues: V

Chief-Editor:

Ubale Amol Baban

30/11/2014

Aarhat Multidisciplinary International Education Research Journal (AMIERJ)

(Bi-Monthly) Peer-Reviewed Journal Vol No III Issues V
ISSN 2278-5655

A SURVEY OF TECHNIQUES IN MINING SOFTWARE REPOSITORY

Dr.Bijendra Agrawal[1] Narendra R Patel[2]

Principal[1] Assistant Professor[2]

[1]College Of Computer & Management Studies,Vadu Ta-Kadi Dist-Mehsana (Gujarat) India

[2]Shree Madhav Insti. Of Comp. & I.T Surat(Gujarat)India

Abstract:

During the software development, Software developers do not perform software-engineering task. In Software repository, source code is available and software developers use these repositories to support their activities. The research discipline of mining software repositories (MSR) uses these extant to understand the software system. So MSR brings together researchers and practitioners to consider methods of using data stored in software repositories to further understanding of software development practices. The main objective of this survey report is to define a research area in MSR and to discuss how MSR techniques are used.

Keywords: *MSR, Source code.*

I. Introduction

During the software development, Software developer do not perform software-engineering task. Source code is one type of software repository from which software developers extract valuable information like issue-tracking repositories [14] and online project-tracking software [6], as well as formal documentation like specifications and manuals and informal communications like emails [10]. The research discipline of mining software repositories (MSR)

uses these extant to understand the software system. So MSR brings together researchers and practitioners to consider methods of using data stored in software repositories to further understanding of software development practices. The main objective of this survey report is to define a research area in MSR and to discuss how MSR techniques are used.

2. Overview of MSR

The purpose of mining software repositories is to use the wealth of information available which is available in software repositories. These information can be very much useful in the software-development process. Information such as issue-tracking repositories, source code, and documents, relationships can be identified, and knowledge of software processes and characteristics can be acquired. This knowledge can be useful in development of system so we improve the performance.

2.1. MSR Data Sources.

Source code is commonly and easily available source of data for MSR. This source code is commonly found in repositories such as Sourceforge, Google Code, Subversion [23], CVS [22], Git [26].

Source like bug repository, mailing list and communication links helps in the analysis of system. So to understand and to know system these information is useful to anyone who related with system.

2.2. Areas of MSR.

(1) Identifying and Predicting Software Quality

(2) Identifier Analysis and Traceability

(3) Clone Detection

(4) Process Improvement and Software Evolution

(5) Social Aspects in Software Development

3. Identifying and Predicting Software Quality

MSR is used to identify quality issues in a software system. MSR develops prediction models to determine how many defects are in the software, and to determine which modules have defects.

Most of the recent literature in software defect analysis examines if there is a correlation between a particular phenomenon, such as a software complexity metric, and defect count, and examines if that phenomenon can be used to predict defect count or defect prone modules. Usually, statistical models are used to identify failure-prone files though some techniques are applicable to individual lines of code or to modules. Zimmermann et al. used the categories \complexity metrics" and \historical data"; complexity metrics included dependency calculations, and historical data in this context included code churn. We use the following categories to broadly classify the current literature in MSR failure counting and failure prediction.

(1) Software metrics, which includes complexity metrics and dependencies
(2) Software evolution, which examines repository histories for changes
(3) Social factors, which examines social interactions, often in relation to technical aspects

As an introduction to the field of defect prediction, Zimmermann et al. provides an overview on preparing software repositories for the analysis of defects. Nagappan et al. also provide a great description of the techniques commonly employed in defect counting and prediction.

3.1. Software Metrics and Quality.

A number of classic metrics have been used extensively for failure prediction, primarily Halstead's complexity measures and McCabe cyclomatic complexity. McCabe tends to be a

relatively good predictor , though recently software metrics have been used in conjunction with other factors. Gall et al. [25] described logical coupling as one such dependency that is constructed by grouping the files together that comprise a bug or feature change to the software. Cataldo et al. [16] proposed workow dependency as a socio-technical dependency that occurs when an issue in the issue-tracking system is reassigned from one developer to another. Zimmermann and Nagappan mapped software dependencies as a dependency graph and applied social network analysis measurements to Windows Server 2003. Nguyen et al. performed a replication of this analysis on Eclipse. Cataldo et al. [16] compared different representations of a dependency to identify failure-prone files. They examined syntactic dependencies based on call graphs; logical dependencies based on relationships between files. SLOCCount [17, 16], CODD [2], tools for metrics estimation (which include algorithms to calculate Halstead's[8] and McCabe's[12] complexity measures), raw count of file sizes (using for instance the wc utility).

3.2. Software Evolution and Quality.

Software evolution techniques use historical data, usually code changes, to predict defects that may occur later in the project. Nagappan and Ball used code churn metrics for predicting defect density in Windows Server 2003. Hassan and Holt proposed the use of a \cache" from the operating system discipline in defect prediction. This was later expanded into the FixCache algorithm.

3.3. Social Factors and Quality.

Cataldo et al. identified a correlation between an alignment between social interaction and software dependencies, called socio-technical congruence, and defect repair times [17]. Nagappan et al. identified that organizational aspects could predict which binaries were prone to post-release failures. A follow-up study on Windows Vista by Pinzger et al. examined the relationship between developers that contributed to the same code module, and determined that social network centrality measures were able to predict failure-prone binaries Bird et al. [12]

examined socio-technical networks, which are networks that represent both contributions and dependencies among developers and binaries. Social network analysis in MSR led to Wolf et al. describing a general technique to apply social networks in software engineering. Cataldo et al. [15] investigated feature changes from a repository and identified that global software engineering in their context was the largest effect contributing to defects.

4. Identifier Analysis and Traceability

Traceability is an important problem in software engineering. Enabling good traceability improves maintenance and program comprehension. One way to approach this problem is through identifier analysis. Two techniques, CamelCase and Samurai, are commonly used for identifier splitting: CamelCase [2], which splits identifiers using underscores, numbers, and alphabetical case changes; and Samurai [21], which further uses substrings to refine identifiers. Both techniques have advantages but it has been identified that they are extremely similar in functional performance [20]. Arnaoudova et al. [5] used identifiers as a basis for their defect prediction model.

5. Clone Detection

Over the years different techniques are proposed to locate clones or fragments which share the same code but may differ in the naming of identifiers. Ducasse for example, proposes a detection technique to locate clones containing a certain amount of identical lines. Baker on the other hand focusses on code fragments in which identifiers, which are likely to change during the duplication process, may differ as long as there is a one to one mapping between the identifiers. Clone detection is the investigation of duplicated sections of source code, usually through the copy and paste function in an editor. Clone detection started to become of interest since it was discovered that 10-15% of code in a software system was copied and pasted [8]. If there is a defect in reused code fragments, then _xing that defect can be problematic. Frequent cloning can also suggest the need for the creation of subroutines that use the cloned fragments. Cloning has

received a lot of attention in source code, with different types of cloning being used such as token-based cloning and abstract syntax tree based cloning [9]. Al-Ekram et al. [1] discovered that source code cloning for the purpose of reuse was not prominent in a number of open-source text editors. Early work by Diessenboeck et al. [19] applied a graph-based algorithm to detect clones.

A parameterized clone detection technique as CCFinder is used. Due to their focus on the detection of similar code fragments, parameterized clone detection techniques are expected to produce the most suitable FACs.CCFinder is a token based detection technique which searches a specially constructed tree for maximal matches. Due to its token based nature, the detection process is not influenced by the code layout. Mockus used word frequency analysis of log messages to not only identify the purpose of changes, but relate it to change size and time between changes as well. Mockus and De Hondt, who both studied change log information, state that a textual description of a change is necessary to understand the real motivation behind a change.

6. Process Improvement and Software Evolution

MSR can identify how source code changes over time during a process. triage a bug [3], identifying the trends of code commits and determining how to file a good bug report. Ratzinger, et al. identified classification techniques that were able to predict refactoring activity. Michael Fischer et al. proposed a heuristic to detect these revisions [8]. Their approach is restricted to merges to the main branch, but it is straightforward to apply it to other branches. Robles, et al. used source code extraction to examine changes over time Ernst and Mylopoulos examined mailing lists to determine if requirement-related discussions became more prominent as software matures,

7. Social Aspects in Software Development

Bird et al. [13] identified that many open-source projects were grouped into \communities", which meant that multiple small groups of developers tended to communicate; his technique allows the identification of community structure in software development projects.

Conclusion:

Using the survey of MSR techniques we can easily choose the best techniques to know and understand the system. There are a number of common techniques through research goals can be achieved, but techniques are applied with some assumption. The analysis techniques used for source code should be applicable to find facts about the system.

References

[1] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In Empirical Software

Engineering, 2005. 2005 International Symposium on, page 10 pp., nov. 2005.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. Software Engineering

[3] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug?

[4] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories.

[5] V. Arnaoudova, L. Eshkevari, R. Oliveto, Physical and conceptual identifier dispersion: Measures and relation to fault proneness.

[6] Atlassian. Bug, Issue, and Project Tracking for Software Development - JIRA [online].

[7] Alberto Bacchelli, Michele Lanza, and Marco D'Ambros. Miler: a toolset for exploring email data.

[8] B.S. Baker. On _nding duplication and near-duplication in large software systems.

[9] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees.

[10] Christian Bird, Alex Gourley, and Anand Swaminathan. Mining email social networks in postgres. In Mining Software Repositories Workshop 2006, ICSE, 2006.

[11] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista.

[12] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures.

[13] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects.

[14] Bugzilla.org. Home :: Bugzilla [online]. 2011. Available from: http://www.bugzilla.org/

[15] Marcelo Cataldo and James D. Herbsleb. Factors leading to integration failures in global featureoriented development: an empirical analysis.

[16] Marcelo Cataldo, Audris Mockus, Je_rey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures.

[17] Barth_el_emy Dagenais and Martin P. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors.

[18] Florian Deissenboeck, Benjamin Hummel, Elmar J□urgens, Bernhard Sch□atz, Stefan Wagner, JeanFran_cois Girard, and Stefan Teuchert. Clone detection in automotive model-based development.

[19] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Clustering support for static concept location in source code.

[20] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identi_ers for software analysis.

[21] Free Software Foundation. CVS - Open Source Version Control [online]. 2006 Available from: http: //www.nongnu.org/cvs/

[22] The Apache Software Foundation. Apache subversion [online]. 2011. Available from:

http:// subversion.apache.org

[23] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In 2007

[24] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history.

[25] git. Git - Fast Version Control System [online]. 2011. Available from: http://git-scm.com