2013 Joint Conference of the 23nd International Workshop on Software Measurement (IWSM) and the Eighth International Conference on Software Process and Product Measurement (Mensura)

Software Repository Analysis for Investigating Design-Code Compliance

Kadriye OZBAS-CAGLAYAN
Software Technologies Research Institute
TUBITAK BILGEM
Ankara, TURKEY

Ali H. DOGRU
Department of Computer Science
Middle East Technical University
Ankara, TURKEY

***Abstract*— Compliance check between code and design is a labor-intensive job to do, since it requires the code to be reverse engineered and checked versus design manually. On the other hand, investigation of the design-code compliance would give some valuable information to software development and maintenance managers. In this study, an approach for employing software repository analysis and text mining techniques to extract and analyze compliance levels of design and code efficiently is presented.**

Keywords—software repository analysis; text mining; design; code; compliance

I. INTRODUCTION

Design is a crucial phase in software engineering since solution to problem defined during analysis phase is created at the design phase. Decisions those constitute the solution is taken during design phase along with selection of best design among the alternatives. On the other hand, developers are supposed to implement the components as specified in the design; it is the input to the coding process performed by one or several (teams of) developers resulting eventually in the source code of the implemented system. Thus, solution is realized in the coding phase and code generated is expected be compliant with design that defines the solution.

When creating / changing the source code, adjusting the design will help developers reduce the complexity. Maintaining high quality of source code and design is a long-term investment and results in a maintainable project. However, in real life, there are cases where developers fail to truly realize the design due to inexperience, lack of knowledge or design becomes obsolete due to software aging (where software quality degenerates over time [1]). Some researchers view problems as non-compliance with design principles [2], violations of design heuristics [3], excessive metric values or lack of design patterns [4], signifying the importance of handling them in the construction and maintenance of software [5]. Non-compliance to design could result in a source code that is unpredictable in terms of code quality.

Compliance check between code and design is a beneficial but labor-intensive job to do, since it requires the code to be reverse engineered and checked versus design manually. Employing software repository mining techniques in compliance-checking between design and code would overcome the difficulties of reverse engineering and manual check processes. On the other hand, investigation of the design-code compliance levels would give some valuable information to software development and maintenance managers.

This study aims to employ software repository analysis and text mining techniques to analyze the compliance levels of design and code.

The remainder of the paper is organized as follows: Section II reviews the related literature. Section III describes our method for design-code compliance analysis. Section IV presents and discusses the results of the case study we conducted to evaluate our design-code compliance analysis. Finally, we conclude with a summary of the contributions of this work and an outline of our future research plans.

II. RELATED WORK

Software engineering researchers have devised a wide spectrum of approaches to uncover relationships and trends from repositories in the context of software evolution. In [6], a tool for identifying structural source code changes from software repositories is introduced. Identified change types are addition / removal of parameter / field / method, hide / unhide/ rename method, pull up / push down of field / method, move attribute / method / class, extract superclass / interface / class.

Data mining version histories is applied in [7] in order to guide programmers along related changes: “Programmers who changed these functions also changed ...” Given a set of existing changes, the mined association rules suggest and predict likely further changes, show item coupling that is undetectable by program analysis, and can prevent errors due to incomplete changes.

In [8], production code and the accompanying tests co-evolution is investigated by exploring a project’s versioning system, code coverage reports and size-metrics.

In [9], relationship between cloning and defect proneness is analyzed and it is reported that great majority of bugs are not significantly associated with clones as well as clones may be less defect prone than non-cloned code.

A way of inferring the evolution of changes at source code level from CVS repositories by combining information

978-0-7695-5078-7/13 $26.00 © 2013 IEEE
DOI 10.1109/IWSM-Mensura.2013.40

retrieval techniques (Vector Space Models) with the Levenshtein edit distance is introduced in [10].

Current state of software repository mining is summarized in [11] as mostly (80 %) focused on source code and bug-related repositories in their analysis of the Mining Software Repositories (MSR) Working Conference and Workshop publications. The analysis of the MSR publications also reveals that documentation repositories (e.g., requirements) are rarely studied, likely due to their limited availability. In the same paper, a new term "Software Intelligence," that offers software practitioners up-to-date and pertinent information to support their daily decision-making processes, is introduced inspired from Business Intelligence which offers concepts and techniques to improve business decision making by using fact-based support systems. It is indicated that the future of MSR should look beyond the coding phase as this phase represents a small portion of the lifecycle of a project.

Among the works about model-implementation compliance checking that has been proposed in the literature, the following are those that explicitly address the problem of checking design against implementation and are applicable in the object oriented domain.

Software Reflexion Model technique tries to help an engineer compare artifacts by summarizing where one artifact (such as a design) is consistent with an artifact and inconsistent with another (such as source) [12].

An approach exploiting edit distance computation and the maximum match algorithm for checking the compliance of object oriented design with respect to source code and support its evolution is presented in [13]. It recovers an "as is" design from the code, compares recovered design with the actual design and helps the user to deal with inconsistencies. The process works on design artifacts expressed in the Object Modeling Technique (OMT) notation and accepts C++ source code. The output is a similarity measure associated to each matched class, plus a set of unmatched classes.

A method that relies on UMLDiff, a UML-structure differencing algorithm, for analyzing the evolution of object-oriented software systems from the point of view of their logical design is presented in [14]. Basically, design-evolution analysis relies on a set of subsequent snapshots of the system source code as the primary input to extract "implemented" logical design.

III. BACKGROUND

In this study, raw data is extracted from a previously finished project which used iterative incremental life cycle and had 4 main iterations. In each of the iterations, new features are added and existing features are improved according to end user feedbacks. Project is implemented using Java programming language and design is modeled using Unified Modeling Language (UML) 2.1 notation. Project work products are baselined at the end of the analysis, design, implementation and system test phases, and each of the iterations consists of these phases.

TABLE I. SIZE OF THE DATA EXTRACTED WRT RELEASES

Phase	Release			
	R1.0	***R1.2***	***R3.0***	***R4.1***
Design	337	347	1144	1144
Code	2344	3035	2767	3118

Our design data is extracted from the class diagrams in the design model baselined at the end of design phase and code data is extracted from the source code baselined for production (at the end of system tests). In order to extract class definitions from source code, a simple string parser is used to match class definitions in the baselined source code. While extracting class definitions from class diagrams in the baselined design model, first Java source code is generated for the design model using the design tool. After generating the source code for the design model, the same simple string parser is used to match class definitions.

Number of class definitions extracted from design and production baselines are given in Table I.

When examining compliance between design and code, only class definitions are taken into account in this study. Analysis of dynamic views (such as sequence diagrams) or relations other than inheritance and realization (such as composition or association) would contribute to this work considerably, however, they are not included within the scope of this article and are identified for future work. Raw data extracted from design and production baselines are pre-processed to extract class definitions with the following attributes:

- Name (String) – name of the class
- AccessLevel (Numeric) – if it is a public / private / protected class
- IsClass (Numeric) – if it is a class or interface
- IsAbstract (Numeric) – if it is an abstract class / interface
- IsFinal (Numeric) – if it is a final class
- Extends (0..n) (String) – classes extended by the class
- Implements (0..n) (String) – interfaces implemented by the class

A class definition in code is accepted to be compliant with design only if all class attributes listed above matches. If any of the attibutes doesn't match then the class definition is counted as non-compliant with design. Data analysis is done using RapidMiner (with Text Processing extension).

IV. RESULTS AND DISCUSSIONS

As a first step of the analysis, exact matching class definitions are extracted. This extraction is done by checking class definitions for exact match using parameters listed above. Results of this analysis are given in Table II (second row). As defined in the previous section, this analysis looks for a perfect compliance, i.e. class definition is compliant only if all attributes match and otherwise it is decided to be non-compliant.

TABLE II. NUMBER AND PERCENTAGE OF CLASSES IMPLEMENTED COMPLIANT WITH DESIGN

Phase	Release			
	R1.0	***R2.0***	***R3.0***	***R4.1***
# of Classes with the Same Name	180	178	847	845
% of Classes with the Same Name	53	51	74	74
# of Classes with Exactly the Same Definition	95	113	342	336
% of Classes with Exactly the Same Definition	4	4	12	11
# of Classes (in Code) Compliant with Patterns Defined in Design	771	1054	2055	2279
% of Classes (in Code) Compliant with Patterns Defined in Design	33	35	74	73

When class names are matched, it is noticed that ~50% – 75% of the classes defined in the design exist in the implementation. When it comes to the percentage of class definitions in the code that exactly matches the definitions in the design, compliance levels get lower (~4% - 11%).

Upon interviewing the project team regarding these results, it is understood that a sample representing a design pattern defined in the design have many realizing class definitions in the code (for example, "all XEventHandler classes should be extending IProjectEventHandler" rule is is defined as a pattern in the design only once, but there are 50+ XEventHandlers in the code those are compliant with this pattern). Here after, regular expressions to represent design patterns are defined to extract design patterns defined in the design and realized in the implementation (i.e. regular expressions to match class definitions like "if the class name is XEventHandler then it should be extending IProjectHandler"). These regular expressions are first checked in the design definition and if there is at least one matching definition, all the matching classes in the code are counted as design-compliant classes.

As seen from the results given in Table II, compliance levels are getting higher as project progresses. One of the reasons behind this outcome is there are very similar but numerous simple classes (approximately % 40 of classes for the first release) such as business rule checkers, warning and error message classes existing in the code but they are not included in the design for the first two releases. Another reason is compliance is defined to fully exist or lack; partial compliance is not considered. On the other hand, there are many class definitions that are partially compliant with their definition in the design. Additionally, when proceeding from R2.0 to R3.0, it is noticed that design is based on the reverse engineered code.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach for software repository analysis to determine the compliance levels of design and code. Class definitions are investigated first for exact match and then for regular expressions those represent the design patterns. Results have shown that percentage of classes those exactly match design definition is relatively low. On the other hand, our pattern approach takes design abstraction level into account and improves compliance levels significantly.

Our approach differs from the work presented in [13] since it focuses on string similarity of fields and methods, where as our approach focuses on string match of attributes in class definition.

In order to improve the results presented in this paper, in addition to "full compliance", "partial compliance" investigation could also be introduced. Partial compliance could be implemented by modifying UMLDiff algorithm presented in [14], where UMLDiff would examine similarity of the names of classes and similarity of classes' relations to check design-code compliance.

Furthermore, compliance definition could be extended to include composition and association relationship as well as inheritance and realization relationships. Additionally, as class diagrams represent the static view of the software design, other views representing the dynamic behavior (such as sequence diagrams, collaboration diagrams, etc.) can also be checked for compliance.

REFERENCES

[1] Parnas D.L., "Software aging," 16th Int. Conf. on Software Engineering, Soronto, Italy, 1994, 279-287.

[2] Martin R. C., Agile software development: principles, patterns and practices, Prentice Hall, 2003.

[3] Riel A.J., Object-oriented design heuristics, Addison-Wesley, 1996.

[4] Gamma E., Helm R., Johnson R., Vlissides J., Design patterns: elements of reusable object-oriented software, Addison Wesley, 1995.

[5] Chatzigeorgiou A., Manakos A., "Investigating the evolution of bad smells in object-oriented code," 7th International Conference on the Quality of Information and Communications Technology, 2010.

[6] Gerlec C., Krajnc A., Heričko M., Božnik J., "Mining source code changes from software repositories," 7th Central and Eastern European Software Engineering Conference in Russia, 2011.

[7] Zimmermann T., Diehl S., Zeller A., "Mining version histories to guide software changes," IEEE Transactions on Software Engineering, pp. 429-445, 2005.

[8] Zaidman A., Rompaey B. van, Demeyer S.; van, Deursen A., "Mining Software Repositories to Study Co-Evolution of Production & Test Code," 1st International Conference on Software Testing, Verification, and Validation, 2008.

[9] Rahman F., Bird C., Devanbu P., "Clones: What is that Smell?," 7th IEEE Working Conference on Mining Software Repositories, 2010.

[10] Canfora G., Cerulo L., Di Penta M., "Identifying Changed Source Code Lines from Version Repositories," 4th International Workshop on Mining Software Repositories, 2007.

[11] Hassan A. E., Xie T., "Software Intelligence: The Future of Mining Software Engineering Data," FoSER 2010, Santa Fe, New Mexico, USA, 2010.

[12] Murphy G. C., Notkin D., Sullivan K. J., "Software Reflexion Models: Bridging the Gap between Design and Implementation," IEEE Transactions on Software Engineering, Vol. 27, no. 4, pp. 364-380, 2001.

[13] Antoniol G., Potrich A., Tonella P., Fiutem R., "Evolving object oriented design to improve code traceability," 7th International Workshop on Program Comprehension, 1999.

[14] Xing Z., Stroulia E., "Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software," IEEE Journal on Software Engineering, pp. 850-868, 2005.