SOFTWARE REUSABILITY BASED ON COHESION AND COUPLING USING HYBRID ALGORITHM

Sudhir Verma
Research Scholar, M.Tech (CSE)
RIMT IET, Mandi Gobindgarh
Punjab Technical University
Punjab INDIA

Charanjit Singh
Assistant Prof. (CSE)
RIMT IET, Mandi Gobindgarh
Punjab Technical University
Punjab INDIA

Abstract:

The object-oriented approach has been the most popular software design methodology for the past twenty-five years. Several design patterns and principles are defined to improve the design quality of object-oriented software systems. In addition, designers can use unique design motifs that are designed for the specific application domains. Another commonly used technique is cloning and modifying some parts of the software while creating new modules. Therefore, object-oriented programs can include many identical design structures. In this paper we are developing an algorithm which help to find the connected modules inside package or from on package to another package so that we can determine the cohesion and coupling in project which help to determine the reusability of modules in the project.

Keywords—Reusability of module

A Introduction

Many software projects contain a significant number of software clones [1], which are duplicated parts of source code or design models. One reason for design-level cloning is the frequent usage of software design principles and design patterns (e.g., GRASP [2], GoF [3]). Furthermore, there are domain-specific patterns [4] that are optimal for a specific application or Thomson Reuters, India is the “fourth most dangerous country” in the world for women, and the worst country for women among the G20 countries. This paper focuses on a security system that is designed solely to serve problem, which allows them to be used repeatedly in a project. In addition, there are unfavorable sources of clones, such as anti-patterns and common design defects. Consequently, a software system can include many identical parts at the design level. In the article ‘Draw Me a Picture’ [5], Booch noted that the hidden patterns (domain-specific patterns) in software are crucial to understanding software architecture and to assessing its quality. Domain-specific patterns are the reused design structures in a specific project domain that are specialized to accomplish similar jobs in a common design form. Detection of these structures will give us the opportunity to improve and publish them for all developers. Theoretically, it is also possible to explore currently unnamed design patterns by examining reused design structures in specific project domains. Studies on software maintenance indicate that more than 2/3 of the total development cost is spent on software maintenance activities [6,7], and more than half of the maintenance cost is spent on comprehension activities [8]. Nevertheless, several real-world evaluations present that the availability of documented design patterns will reduce the cost of program comprehension for object-oriented systems [9,10]. Additionally for these reasons, it is important to discover reused design patterns. The comprehension of software architecture also plays a key role in refactoring processes, which constitute the reactive part of maintenance tasks. Refactoring improves the internal design structure of software by preventing the production of poor quality products.

Identifying these types of identical design structures (i.e., non-standard design patterns and common design defects) could provide a significant advantage in terms of reducing the cost of maintenance; the reason is that the most commonly-used structures in software design are the best places to look for refactoring opportunities because they affect multiple parts of the design. For example, non-standard structures that are similar to design patterns might be modified to conform to standard forms, and common design defects can be quickly identified, which allows them to be fixed in multiple areas at once. In addition, frequently repeated identical design structures are usually the most reusable parts of the design; these parts can provide good candidates for additional use in future designs. Another source of the clones is the replicated code due to copy–paste activities. Mostly, developers modify these replicated parts separately to allow their source code to change, but the design remains same [11]. The code quality could decrease if developers apply a bug fix to one structure but fail to apply the same correction to its copies. With the help of our approach, copy–paste type design structures can be detected even if their source code is modified. These replicated structures can be combined into a single library entity, to be used efficiently in different parts of the current project or in future projects. This approach will also avoid inconsistent bug fixes. The area of clone detection is considered to be an important part of several software engineering tasks [12]. Finding repetitive design matches could help developers and architects when evaluating reused design patterns, refactoring duplicated parts, understanding the program architectures and detecting plagiarism. In this paper, we propose a graph mining-based approach to detecting identical parts in an object-oriented software architecture. The proposed approach contains three main steps. In the first step, the AST (abstract syntax tree) of the source code of the system is analyzed and the UML-based design level of abstraction is created. Based on this abstraction, we construct a software model graph, in which classes, interfaces and templates of software constitute the vertices, and the relations between them form the directed edges. According to the importance of the relation type, we assign weight values to the edges of the graph. In the next step, we apply a graph partitioning algorithm to divide the directed and weighted software model graph into small pieces. Finally, in the last step, a sub-graph mining algorithm is applied to discover identical design structures in the generated software model. Because the scope of our study is primarily focused on the detection stage, analyzing and automatically classifying these structures could be an interesting topic for future studies. However, we also present several evaluations with a manual classification to find useful and meaningful structures from open-source and industrial projects that could inspire future studies. The remainder of this paper is organized as follows: Background and related work are presented in the next section. The graph representation and definitions are given in Section 3. The identification process is detailed in Section 4. In Section 5, the results obtained from exemplary projects are presented. In Section 6, we discuss critical parts and the efficiency of our approach, and the last section concludes the paper.

A. Structural Composition

Structural composition approaches build a design by gluing pattern structures that are modeled as class diagrams. Structural composition focuses more on the actual realization of the design rather than abstractions as role models. Behavioral composition techniques, such as roles [18,20] leave several choices to the designer with less guidelines on how to continue to the class design phase. This probably creates more confusion than flexibility. Therefore, we advocate a structural composition approach with pattern class diagrams. a) Software Composition using Design Components. Keller and Schauer [10, 22] address the problem of software composition at the design level using design components. Their project, called Software desirable Properties into the design of ObjectOriented Large-scale software systems (SPOOL), focuses on the problem of adapting software systems in large telecommunication companies due to rapid changing requirements. Their approach and ours share the same objective of creating software designs that are based on well-defined and proven design patterns packaged into tangible, customizable, and composable design components. Keller defines the evolution of a pattern in a design boundaries around pattern participants. Such a technique, also used in [13], is cumbersome and does not support a high-level view of the design.

B Design pattern definition

In general, a design pattern is described in a template consisting of two sections, Problem Domain and Solution Domain. The Problem Domain describes the problem context where the pattern can be applied. Analogously, the Solution Domain describes the structure and collaborations of the pattern solution being applied to the problem. A design pattern consists of Pattern Name section, Intent section (description ofits problem), Motivation section (a scenario that illustrates a design problem), Applicability section (the situations in which the design pattern can be applied), Structure section (a structure of the participants in the Solution Domain), Participants section (the classes and/or objects participating in the Solution Domain), Collaborations section (collaboration diagrams between solution participants), Consequences section (the results and trade-offs of applying the pattern), Implementation section, and the Related Patterns section[1]. In this paper, only the Solution Domain of each design pattern document including Structure, Participants, Collaborations, and Implementation sections is used to detect automatically design patterns.

C. Dynamic cohesion measurement

In object-oriented systems, attributes and methods are the basic elements of an object or a class. A well-defined theoretical framework that formally defines these elements and depicts the relationships among the elements is the precondition of a welldefined cohesion measure. Here, a novel theoretical framework is proposed for characterizing elements and dependence relationships among elements of an object or class. This framework is used to describe relationships of four types: (i) write dependence relation between attributes and methods, (ii) read dependence relation between methods and attributes, (iii) call dependence relation among methods, and (iv) reference dependence relation among attributes. The proposed measures take into account two types of access relationships between methods and attributes, i.e. read access relations and write access relations between methods and attributes. If a method having some logical error writes an attribute, then the value of attribute may also be incorrect. Thus, value of the attribute is dependent on the behaviour of the method during write access relationship between methods and attributes. Similarly, if a method reads an attribute that has incorrect value, then behaviour of the method may also be erroneous. Though, if method has a logical error, the value of attribute will not be affected by the method reading it. This fact states that the behaviour of method is dependent on the value of attribute during read access relationship between methods and attributes [1]. The proposed metrics account for inheritance and polymorphism present in object-oriented software. During dynamic cohesion measurement, we treat class (including inherited features) as a single semantic concept. Thus, set of attributes and set of methods of a class (formally defined in the next section) include set of inherited attributes and set of inherited methods, respectively. The concept of polymorphism is relevant only in method invocation type of connections. Since, cohesion is being measured for an object at run-time; polymorphic method invocations are accounted for automatically instead of static method invocations. s

D Coupling and Cohesion

The term coupling is used to measure the relative inter- dependency between various classes as one class has the link with another class. While on the other hand cohesion is defined as the strength of the attributes inside the class which means how the attributes are linked inside the class. Coupling is always correlated with cohesion in such a way as if coupling is high then cohesion is low and vice versa. One can say that a class is highly coupled or many dependent with other classes, if there are many connections and loosely coupled or some dependent with other classes if there is a less connections. The coupling is decided at the designing phase of the system, it de- pends on the interface complexity of the classes. There- fore, the coupling is a degree at which a class is con- nected with other classes in the system.

Let us now describe the cohesive class which can per- form a single task within the software procedure. It re- quires little interaction with other procedures that are used in other parts of a program. Cohesion gives the strength to the bond between attributes of a class and it is a concept through which capture the intra-module with cohesion. Therefore, cohesion is used to determine how closely or tightly bound the internal attributes of a class to one another. Cohesion gives an idea to the designer about whether the different attributes of a class belong together in the same class. Thus, the coupling and cohe- sion are related with each other.

VI. Conclusion and future work

Software Reusability play an important role in Software companies which help to the company to save time for development of new project with specific timeline.But finding the reusability modules is based on highly managerial skills generally small company cannot afford high skills people fo this we have developed the Tool which help to find the reusability of module in existing software

References

[1] G. Gui and P. D. Scott, "New Coupling and Cohesion Metrics for Evaluation of Software Component Reusabil- ity," Proceedings of the 9th International Conference for Young Computer Scientists, Zhangjiajie, 18-21 November 2008.

[2] I. Vanderfeesten, H. A. Reijers and W. M. P. van der Aalst, "Evaluating Workflow Process Designs Using Co- hesion and Coupling Metrics," Computers in Industry, Vol. 59, No. 5, 2008, pp. 420-437. doi:10.1016/j.compind.2007.12.007

[3] T. M. Meyers and D. Binkley, "An Empirical Study of Slice-Based Cohesion and Coupling Metrics," ACM Trans- actions on Software Engineering and Methodology, Vol. 17, No. 1, 2007, Article No. 2.

1. [4] Y. J. Jeong, H. S. Chae and C. K. Chang, "Semantics Based Cohesion and Coupling Metrics for Evaluating Understandability of State Diagrams," IEEE 35th Annual Computer Software and Applications Conference, IEEE Computer Society, Washington, 18-22 July 2011.

[5] M. E. Fayad and A. Altman. An introduction to software stability, Communications of ACM, Vol.44, 2001; 9: 95-98.

[6] M.E. Fayad, M. Cline. Aspect of Software Adaptability. Communications of ACM, Vol. 39, 1996; 10:58- 59.

[7] Dijkstra E. W. A Discipline of Programming. England-wood Cliff, New Jercy: Prentice-Hall; 1976.

[8] Parnas D. On the Criteria to be Used in Decomposing Systems into Modules. Communications of ACM, Vol. 15, 1972:12:1053-1058.

[9] Lopes C. V. and W. L. Hursch. Separation of Concerns. College of Computer Science, Northeastern University, Boston; 1995.

[10] R.E. Filman et al. Aspect-Oriented Software Development, Addison-Wesley; 2004.

[11] Abreu, F.B., Melo, W., 1996. Evaluating the Impact of Object-OrientedDesign on Software Quality. In: Proceedings of 3rd InternationalSoftware Metrics Symposium (METRICS'96), IEEE. Berlin. pp. 90–

[12] O. Hummel and C. Atkinson, Using the Web as a Reuse Repository, Reuse of Off-the-Shelf Components, Lecture Notes in Computer Science, vol. 4039, Springer, 2006, pp.298-311.

[13]A. Ampatzoglou, K. Apostolos, G. Kakaronzzos, and I. Stamelos, An Empirical Evaluation on the Reusability of Design Patters and Software Packages, Journal of Systems and Software, vol. 86, Dec. 2011, pp. 2265-2283.

[14]G. Kakarontzas and I. Stamelos, Component Recycling for Agile Methods, Sev-enth International Conference on the Quality of Information and Communications Technology (QUATIC), 29 Sept. 2010 – 2 Oct. 2010, pp.397-402.