Software Stability in Software Reengineering

CHIA-CHU CHIANG
Department of Computer Science
University of Arkansas at Little Rock
2801 South University Avenue
Little Rock, Arkansas 72204-1099, USA
cxchiang@ualr.edu

Abstract

Legacy systems won't evolve well in today's modern computing environments without reengineering. Unfortunately, most reengineering projects are only concerned about whether the systems can be seamlessly integrated into the environments and usually ignore the quality in the improvement of the legacy systems. We all agree that it would be better to observe both integration and quality improvement implemented in the reengineering process. Software stability makes this possible in the reengineering process so that the new systems can run in a heterogeneous computing environment but also be stable enough to reduce the maintenance costs and efforts. In this paper, a connection between the stability modeling and reengineering process for legacy system is described. A preliminary study on techniques for building a better stable system will be presented although the techniques are still not very promising yet. We will also discuss the issues and challenges of applying software stability into the legacy reengineering process.

Keywords: Forward Engineering, Legacy Integration, Reverse Engineering, Software Engineering, Software Maintenance, Software Stability, and Software Reengineering

1. Introduction

Legacy systems were designed to run on a centralized system. Usually, they are monolithic and have gone through a long period of changes that are difficult to maintain. Due to the emerging technologies of computing, companies expect their legacy systems to be integrated into a heterogeneous computing environment to respond to the rapid changes of daily business operations. There are three different approaches: rewriting, wrapping, and reengineering for legacy integration. However, many companies are unwilling to rewrite their legacy systems from scratch due to high cost. Most of the companies use the wrapping approach to integrate their legacy systems. Unfortunately, the wrapping approach does not target the quality improvement of the systems. As a result of reengineering, companies are required to continuously invest money in their legacy systems for maintenance. From a long-term investment point of view, the wrapping approach won't save the costs for the companies. Reengineering is an alternative approach to renovating legacy systems for legacy integration. However, reengineering won't help future maintenance if the core of the systems is not changed. Constant changes will still be needed if the unstable parts of the systems are not reengineered. However, software stability can be built in the reengineering process to make legacy systems more maintainable for changes that have less impact on the stable modules of the systems.

In this paper, we will look into the integration of software stability models using reengineering. The key techniques of reengineering, issues and challenges of integrating software stability into the process, and possible techniques that can be used to make legacy systems more stable after reengineering will be presented in this paper. What this paper is going to present is still in a preliminary step. However, this paper will bring attention to a connection between the stability modeling and reengineering process. The work presented will be the beginning of a promising research.

1-4244-1500-4/07/$25.00 ©2007 IEEE

2. Software reengineering and software stability

Software maintenance is an inevitable process due to program evolution [19]. Adaptive maintenance [20] is a process used to adapt software to new environments or new requirements due to the evolving needs of new platforms, new operating systems, new software, and evolving business requirements. Applications must evolve rapidly to provide quick response to changing business requirements without compromising system quality. Reengineering [7], also known as renovation and reclamation, begins with the analysis of a system and adaptation of the system based on the knowledge obtained from the analysis to new environments. Basically, reengineering can be viewed as reverse engineering followed by forward engineering. Reverse engineering is a process of analyzing a system for program understanding. Forward engineering is a traditional process of software development including requirements and specification, design, coding, and testing. Reengineering as shown in Figure 1 utilizes the techniques of reverse engineering to obtain the knowledge of the systems and then use this information to help build a system in a new form by following the phases of forward engineering.

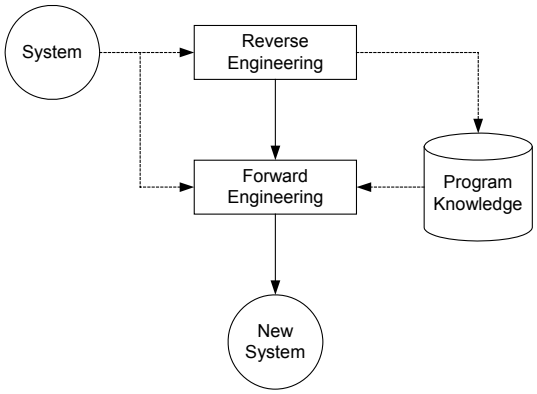

Figure 1: Reengineering process

Software stability reduces the impact of software changes by dividing a system into stable modules and unstable modules. Changes made to unstable modules for software evolution should not impact the stable modules. Although there is no silver bullet, the software stability model intends to minimize the maintenance efforts due to changes. Several studies have investigated the impact of software development on long-term maintainability. Most of the studies are focusing on forward engineering and very few have discussed the applicability of software stability in reengineering legacy systems. Existing reengineering techniques are focusing on the conversions of legacy systems into new modern systems and usually ignore the quality of the new systems. Software stability will bring us attention to the building of better stable systems in connection with the reengineering process.

2.1. Software stability in the requirements and specification phase

Fayad and Altman introduce software stability and discuss how to deal with software stability in the front-end of the software development process. A system is modeled into enduring business themes, business objects, and industrial objects. A set of criteria for determining candidates of enduring business themes, business objects, and industrial objects are also discussed in terms of stability over time, adaptability, essentiality, intuition, explicitness, commonality to the domain, and tangibility [10-12]. For software engineers who are not familiar with stability will feel difficult for determining the objects for stability. Fayad subsequently presents heuristics to help determine enduring, business, and industrial objects in addition to traditional top-down and bottom-up approaches.

Hamza [17] presents an approach to designing a system that is less likely to change over time using the software stability model [10] and formal concept analysis [13]. In the requirements and specification phase, a set of functional requirements, non-functional requirements, and a set of classes that are classified as enduring business themes, business objects, and industrial objects are identified. From the functional, non-functional, and classes, a set of use-case scenarios are identified and specified. With the classes and use-cases, a formal context, which is represented as a two-dimensional table where each class forms a row and each use-case scenario forms a column, of the system is created. The intersection of the i^{th} row and j^{th} column indicates that the class in the i^{th} row realizes the corresponding use-case scenario in the j^{th} column. A formal concept in the formal context is a pair of classes with respect to a set of use-case scenarios. Obviously, a set of concepts can be identified from the formal context that needs to be filtered. Hamza provides a set of rules to filter out irrelevant concepts. Next, a set of concerns are identified from the functional and non-functional requirements of the system and realized by a set of use-case scenarios. From the concerns, we can compute the correlation of the concepts and the concerns that the concepts realize corresponding to the use-case scenarios. Hamza provides a set of metrics to

assess the correlation between the concepts and concerns. Irrelevant concepts are eliminated and a concept lattice is created from the resulting concepts that indicate the inheritance relationship of the concepts with respect to the use-case scenarios. Since we already know which classes will realize the concepts and which concepts will realize the concerns, we therefore know which classes will realize the concerns. As a consequence, concerns about unstable concepts are designed into industrial objects and enduring business themes and business objects should realize the concerns about stable concepts.

Bush and Finkelstein [3] present an approach to assessing requirements stability using scenarios. To conduct requirements stability analysis, a goal list and a set of scenarios need to be developed first. The scenarios are then analyzed for stability against the goal list. The process is similar to code walkthrough and guidance is provided to aid the stability analysis.

2.2. Software stability in the design phase

Design can be conducted in two levels: architectural design and low-level design. Architectural design specifies a system in terms of computational components (modules, functions, procedures, or subsystems) and interactions among those components [21]. Architectural stability refers to the extent to which a software architecture is flexible enough to respond to changes due to the changes in requirements.

Bahsoon and Emmerich [2] present a model to evaluate the architectural stability using options theory. The theory provides an analysis paradigm that help to find an architecture that maximizes the investment value relative to future changes in software evolution.

Tonu et al. [22] present a metric-based approach to evaluate software architecture for software stability. Architectural stability can be evaluated using growth rate, change rate, cohesion, and coupling of the system.

Kelly [18] conducted an experiment where a set of design metrics are chosen and examined how they have potentially contributed to long-term maintainability. The findings of this study help to select the right design metrics for assessing the design stability in the design phase.

Elish and Rine [8] conducted a similar study on object-oriented design stability. The experimental results concluded that Weighted Methods per Class, Depth of Inheritance Tree, Coupling between Object Classes, Response for a Class, Lack of Cohesion in Methods metrics are negatively correlated with the logical stability of classes. No correlation was found between Number of Children and the logical stability of classes.

Elish [9] examines design patterns on the stability of classes. The results indicate that there is a positive impact of the design patterns [14] such as adapter, bridge, composite and façade) on the stability of class diagram. Grosser et al. [15-16] present a model for predicting design stability of Java classes.

3. Issues and challenges

Reengineering begins with reverse engineering. Unfortunately, many legacy systems do not have documents but code only. It is also not unusual to see no source code in a legacy system. Therefore, most of reverse engineering tools count on source code for analysis. No reverse engineering tool can fully understand the source code due to the inherent complexity of the legacy system itself. Humans usually need to get involved to fill in the blanks that the tools leave out. The concepts, tools, techniques, case studies, risks, benefits, and research possibilities of reengineering can be found in the reference [1].

The next step is to build a new system following the traditional software development process which stability can be kicked in to help build a better stable system. Although several methods have been presented in Section 2, however, I would like to see these methods to be more practiced in large projects. In addition, many studies right now are still centering on the understanding of design metrics for contributing stability of systems. There is no doubt in the near future, we will see the results reflected into design guideline, principles, and techniques for a better stable design. Also, very few studies are investigating the impact of software testing techniques for stability.

One benefit of software reengineering is to reuse the code in legacy systems. Code from the legacy system is extracted into classes for software reuse. Program slicing [24] is a technique to extract the code according to the concepts identified in a concept lattice. Chiang [6] presents a method of code extraction from legacy systems. However, as legacy systems were designed, there are no concepts of classes designed into the systems. Code may be dispersed in the programs and need to be collected into corresponding classes according to the design document. Secondly, code extracted might be for industrial objects only. To complete the system, new code for enduring business themes and business objects need be written from scratch. Thirdly, the code extraction requires human interaction during the process [23], even with the aids of tools.

4. Summary

Fayad argued that a software stability model can help design a more stable system with likely less reengineering. Currently, most of reengineering project focus on the implementation of the new system that preserves the original functionally but ignore the impact of quality on long-term maintainability. In this paper, we bring in the software stability into the reengineering process. Techniques for building a better stable system were presented although they are still under development. Once the techniques are more concrete and mature, we believe using these techniques, a legacy system can be reengineered into a more stable system that leads to long-term success in software evolution.

What we presented in this paper is still a preliminary study. However, we believe that the presented work will bring attention to the connection between the software stability modeling and software reengineering process. In the near future work, we will look into our software reengineering tool [4-5] for software stability. What kind of information the reengineering tool is required to produce so the information can be used to help partition a legacy system into stable and unstable modules for forward engineering.

5. References

[1] R. S. Arnold, *Software Reengineering*, IEEE Computer Society, 1994.

[2] R. Bahsoon and W. Emmerich, "Evaluating Architectural Stability with Real Options Theory," *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004, pp. 443-447.

[3] D. Bush and A. Finkelstein, "Requirements Stability Assessment Using Scenarios," *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*, 2003, pp. 23-32.

[4] C.-C. Chiang, "Reengineering Enterprise Systems for Y2K Compliance," *Proceedings of the 23rd Annual Conference, The Chinese-American Academic and Professional Association in Southeastern United States (CAPASUS 99)*, July 9-11, 1999, Atlanta: Georgia, USA, pp. 29-38.

[5] C.-C. Chiang, "Encapsulating Legacy Systems for Use in Heterogeneous Computing Environments," *Information and Software Technology*, Vol. 43, No.8, July 2001, pp. 497-507.

[6] C.-C. Chiang and C. Bayrak, "Legacy Software Modernization," *Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics (IEEE SMC 2006)*, October 8-11, 2006, Taipei: Taiwan, ROC, pp. 1304-1309.

[7] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery," *IEEE Software*, Vol. 7, No. 1, January 1990, pp. 13-17.

[8] M. O. Elish and D. Rine, "Investigating of Metrics for Object-Oriented Design Logical Stability," *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, 2003, pp. 193-200.

[9] M. O. Elish, "Do Structural Design Patterns Promote Design Stability?" *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 2006, pp. 215-220.

[10] M. Fayad and A. Altman, "An Introduction to Software Stability," *Communications of the ACM*, Vol. 44, No. 9, September 2001, pp. 95-98.

[11] M. Fayad, "Accomplishing Software Stability," *Communications of the ACM*, Vol. 45, No. 1, January 2002, pp. 111-115.

[12] M. Fayad, "How to Deal with Software Stability," *Communications of the ACM*, Vol. 45, No. 4, April 2002, pp. 109-112.

[13] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[15] D. Grosser, H. A. Sahraoui, and P. Valtchev, "Predicting Software Stability using Case-Based Reasoning," *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002, pp. 295-298.

[16] D. Grosser, H. A. Sahraoui, and P. Valtchev, "An Analogy-based Approach for Predicting Design Stability of Java Classes," *Proceedings of the 9th International Software Metrics Symposium (METRICS'03)*, 2003, pp. 252-262.

[17] H. S. Hamza, "Separation of Concerns for Evolving Systems: A Stability-Driven Approach," *Proceedings of*

Workshop on Modeling and Analysis of Concerns in Software (MACS'05), 2005, pp. 105-110.

[18] D. Kelly, "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion," *IEEE Transactions on Software Engineering*, Vol. 32, No. 5, May 2006, pp. 315-329.

[19] M. M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, London.

[20] N. F. Schneidewind, "The State of Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, pp. 303-310.

[21] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[22] S. A. Tonu, A. Ashkan, and L. Tahvildari, "Evaluating Architectural Stability Using a Metric-Based Approach," *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006, pp. 261-270.

[23] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Human Factors in Extracting Business Rules from Legacy Systems," *Proceedings of 2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE SMC 2004)*, 2004, pp. 200-205.

[24] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, pp. 352-357.