Challenges and Discussion of Software Redesign

Marija Katić
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
marija.katic@fer.hr

Krešimir Fertalj
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
kresimir.fertalj@fer.hr

ABSTRACT

Software design complexity is increased while software is developing and therefore a management of the design complexity is an important issue. In order to accomplish this task various methods have been developed so far. Some methods propose crucial places where the software might be too complex leaving redesign to be accomplished manually. Other methods try to automate the redesign process as much as possible. This paper presents main definitions and terms concerning software redesign, current research in this area and challenges that might be potential candidates for the further research.

Key Words: software redesign, refactoring, reengineering, challenges

1. Introduction

Software evolves and its complexity is increased over the time. When project fails for reasons that are primarily technical, the main reason is often uncontrolled complexity [4]. Therefore it is crucial to maintain software design as simple as possible in order to ensure preservation of software quality. On the one hand, software redesign means improvement of structural integrity (internal structure), in other words restructuring (Figure 1). According to Chikofsky and Cross [3] restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the system's external behaviour (functionality and semantics). When this transition is done in object oriented systems, it usually requires changing the abstractions built in classes and the relationships among them, and in this case it is referred to as refactoring [8]. On the other hand, when it is observed from the more general view, which means changing software system, redesign is designated as reengineering. Therefore the term of redesign is closely related to the reengineering and refactoring terms and its definition is somewhere in between reengineering and refactoring or restructuring. Reengineering is a wider term than restructuring and it involves restructuring which shows how these terms are interrelated.

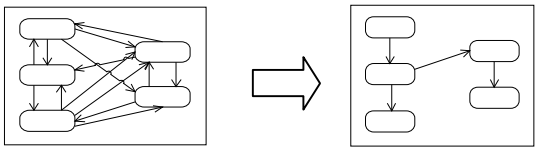

Figure 1

This paper gives more attention to the redesign as refactoring and it is organized in the following way. First of all, it is emphasised that the redesign can be performed on all software artefacts. Main redesign activities and the current research are given in the further text. Finally, there are challenges that should be concerned when thinking about further research.

2. Software artefacts

All software artefacts in software development, such as documentation, design models, database schemas, source code, or test cases can be exposed to the redesign process. Only those related to programs (source code) and design models are considered in this paper. With respect to programs, refactoring is referred to the source code and transformations of its structure. Those programs that are not written in an object-oriented language are much more difficult to restructure because data flow and control flow are tightly interwoven [2]. At design level, refactoring can be performed in the same way as program refactoring, but it is referred to as model refactoring. For instance Astels [10] proposes using an UML tool as an aid in finding parts that need redesign and performing appropriate redesign method.

3. Examples of source code redesign methods

M. Fowler gives a list of refactorings that can be useful for developers to improve the design of their code, in other words source code redesign methods [1]. Some representative examples are as follows:

Composing methods

- Extract Method. Extraction of a piece of code into a separate method.
- Replace Temp with Query. Replacement of references to the temporary variable with the method calls. It facilitates method extraction.

Moving features between objects:

- Move Method. Moving method to another class when a class has too much behaviour or when classes collaborate a lot and are too highly coupled. Moving method is the bread and butter of refactoring.
- Extract Class. Creation of new classes with methods and fields from old classes.

Organizing data:

- Replace Type Code with Class. Replacement of state constants with type safe Enum.
- Change Value to Reference. When a class has many equal instances then is better to replace it with a single object or turn the object into a reference object.

Simplifying Conditional Expressions:

- Replace Conditional with Polymorphism. Whenever it is possible it is good to avoid writing an explicit conditional when object's behaviour varies depending on their types.
- Decompose Conditional. Method extraction from conditional statements.

Big Refactorings:

- Convert Procedural Design to Objects. Conversion of a procedural code into object oriented code.
- Extract Hierarchy. If a class is doing too much work, maybe it is good to extract special cases into their own subclasses.

4. Redesign activities

Redesign is a method applied on an existing part of software and therefore it is crucial to identify places and situations when it should be applied. When considering source code, M. Fowler [1] calls these places *bad smells* (duplicated code, long method, large class, and so on). The main two steps in the redesign process are identification of places that are in need for redesign – bad smells identification and determination of appropriate redesign method – restructure execution.

5. Current Research

The term refactoring was first introduced in literature by William F. Opdyke in his PhD dissertation in 1990 [11], where he defined it as a program restructuring operations. His focus was on the automation of those operations. The next breakthrough was when Roberts, Brant and Johnson [12] built the real first refactoring tool for Smalltalk, Smalltalk Refactoring Browser. However, the main interest in this area has started in the late '90s with the book of Martin Fowler [1] and with the agile movement. Fowler defines refactoring as a changing of the software internal structure in such a way that preserves its observable behaviour [1]. At first research was focused at finding individual problems and applying transformations manually [9] [13]. Over the time, the main focus was moved to the automation of refactoring and to the development of tools capable to perform an identification of program parts that need refactoring and proposals and application of refactorings [14]. All modern integrated development environments (IDEs) implement refactoring support at the base level (Move Method, Extract Method, Rename Class …). Most of them provide even advanced refactoring support. For example, when with only one keyboard shortcut it is possible to invoke contextual availability-checking system to determine which refactorings are currently available [24]. More recently refactoring has been applied on more abstract levels, but not only on the source code. More research have appeared with respect to refactoring at the design level [21] [22], especially in terms of UML models, after survey on software refactoring had published [2] [21]. For automatically performing redesign in two steps, bad smells identification and refactoring execution, a set of formalisms, techniques and tools is needed. Therefore in the further text all of them are discussed.

5.1. Formalisms and techniques

There are varieties of formalisms to deal with redesign methods. Some of them are graph transformations, software metrics, program analysis, clustering and Meta modelling.

Graph transformation. Software artefacts can be represented as graphs and refactorings as transformation rules. Bottoni, Parisi-Presicce and Taentzer [15] maintain consistency of code and specification during refactoring by describing refactoring by distributed graph transformation. Van Eetvelde and Janssens [16] use a hierarchical representation of object-oriented programs.

Software metrics. Software metrics can be used for measuring quality of software before and after software redesign. O'Keeffee and Cinneide [6] tend to improve the structure of inheritance hierarchies by treating object oriented design as a combinatorial optimization of metrics. Redesign in this case is a search through the space of alternative designs for those that are superior to the original, judging by the metric values [6].

Program analysis. Program analysis is a technique that can help in discovering bad smells, no matter whether it is static or dynamic. Static code analysis plus automatic refactoring equals painless coding [17]. Dynamic program analysis is useful when not all desired preconditions of a refactoring can be statically computed in a reasonable time and computation effort [2].

Clustering. Clustering is a data mining activity. It is an unsupervised learning of a hidden data concept where data are distributed into groups called clusters and each group consists of objects that are similar between themselves and dissimilar to objects of other groups [20]. Czibula and Serban use clustering in order to recondition

a class structure of a software system [5] [19].

Meta modelling. Meta modelling is the mapping of specification concepts onto entities, relations and attributes of a specific domain [21]. It enables redesign that does not depend on implementation language. On the other hand, there is an approach to deal with the transformation of models from a source model to a destination model without changing the observable behaviour [23].

5.2. Tools

As it has already been stated above, redesign can be performed manually, but the main focus in researches is the redesign process automation and the development of tools that support such a process. Redesign is closely connected with testing. Although one can say that for example a source code redesign belongs to the implementation phase, tests are needed to ensure that the behaviour is not changed. It can be said that the redesign without testing does not have sense. Therefore in the context of the redesign testing tools also should be considered. Generally tools can be divided into semi-automated and fully-automated. First refactoring tool Refactoring browser is an example of semi-automated tool [12] and approach that can be stated as fully is developed in [7]. Fully-automated approach is an add-in for Microsoft Visual Studio whose developers [24] were focused on the most common barriers between programmers and refactoring tools: discoverability, lack of trust and productivity. For example to enhance discoverability, they have added background code analysis and highlight mechanism to highlight code smells where powerful, but less well-known refactorings are available [24]. Project Analyzer and Visustin are examples of semi-automated tools [25].

6. Current challenges

Although lots of problems have been realized, also there are lots of challenges to deal with in the area of software redesign:

Challenge 1: Redesigning an existing software system is actually the start of a new project. When it is necessary to deliver a business value as soon as possible as well as to improve the existing system, agile development seems to be useful method to apply. Although there are some experiences [26], there is no generic approach to accomplish this task certainly successfully, especially when redesigning of big legacy software is done in the agile way.

Challenge 2: At the lower levels there are many successful researches on how to perform low-level refactoring [9] [12] [13], however it is not the same with respect to a high-level refactoring [19]. Those are sequences of refactoring rules that consist of several low-level refactorings, for example in order to support the implementation of certain design pattern. They are especially useful when it is about performance improvement, higher modularity and so on. The challenge is to explore how the design will look if such rules are applied or what is the appropriate redesign degree for the performance that seems to be sufficiently improved. Also there is a lack of such tools. It would be good to have a generic solution that is independent on implementation language and that can estimate to which extent it is the most useful to apply redesign methods for the certain benefit.

Challenge 3: With respect to the model refactoring, for example refactoring of class diagrams has been investigated by various researchers [10] [21] [22] [23], but certainly more work is needed for refactoring of behavioural models, especially because of an ambiguity of UML.

Challenge 4: If the redesign is performed at the lower levels such as source code or models, metrics can be applied to estimate a given value [18]. On the other hand, if it is performed more generally, on the whole software system, the challenge is to estimate the given value of redesign. Related problem is maintaining consistency between models, source code, documentations as well as other software artefact.

Challenge 5: It is a challenge to make a categorization with respect to redesign formalisms and techniques that are best suited for a certain purpose. The purpose can be related to project types (business, scientific …), application types (web, desktop …) as well as project size or some other type of software that is considered useful.

Challenge 6: The increase of refactoring tools should not be questionable at all. However, a good comparison of those tools integrated into IDEs [12] [17] [24] [27] as well as standalone tools [25][27] is needed in order to reveal their actual value. Therefore it is necessary to search for them and identify whether and how they can be combined to act more efficiently. We think that each of them should be tested on all different aspects that actually support (different implementation languages, different architectures, different platforms …). The search should consider commercial and research tools, investigate their directions, reliability, configurability, scalability and propose their possible combination.

Challenge 7: Programming paradigms is not changed every day, but new principles are evolved, we presume to say, every day. Therefore, we believe, it could be useful to have a solution that can learn from new principles, using some data mining concept, and propose places in software that should be redesigned. In accordance with this idea, the real challenge would be the redesign conduction.

Challenge 8: Software security is an important issue that should be considered when refactoring activities are performed, especially in the context of web applications. There are identified some refactoring transformations that could affect the security of an existing software [28]. It is needed to ensure that whenever the software is redesigned the security of software must not be undermined and disorganized. Moreover, redesign is supposed to reduce software vulnerabilities. Research with respect to this topic is still in infancy.

7. Conclusion and Future Work

This paper briefly presents the software redesign process and methods that are used in achieving that process. Some of the used formalisms and techniques are also briefly described. Research has shown that redesign process had been applied on more abstract levels and not only on the source code. Although there are lots of different approaches, current challenges are stated showing the places that need more research. Our future work will continue in the direction of improvements of the source code redesign methods. We plan to search for the differences between the redesign of legacy code and the redesign in the agile development environment. Our first step is to face with the challenge 6.

References:

[1] M. Fowler, and K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design o Existing Code, Addison Wesley, 1999.

[2] T. Mens and T. Tourwe, "A Survey of Software Refactoring", IEEE Transactions on Software Engineering, IEEE Press, USA, 2004, pp.126-139.

[3] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy", IEEE Software, 1990, pp. 13–17.
[4] S. McConnell, Code Complete, Second Edition, Microsoft Press, 2004.
[5] I.G. Czibula and G. Serban, "Hierarchical Clustering for Software Systems restructuring", INFOCOMP Journal of Computer Science, Brazil, 2007, pp.43-51.
[6] M. O'Keeffe and M. O Cinneide, "Towards Automated Design Improvement Through Combinatorial Optimization", Workshop on Directions in Software Engineering Environments, Scotland, UK, 2004.
[7] M. O Cinneide, "Automated Application of Design Patterns: A Refactoring Approach", PhD thesis, University of Dublin, Trinity College, 2001.
[8] W. F. Opdyke, "Refactoring object-oriented frameworks", PhD dissertation, University of Illinois at Urbana-Champaign, 1992.
[9] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code", Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, USA, 1999, pp. 109–118.
[10] D. Astels, "Refactoring with UML", Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, Italy, 2002, pp. 67-70.
[11] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems" Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, New York, September 1990.
[12] D. Roberts, J. Brant and R. E. Johnson, “A refactoring tool for Smalltalk”, Theory and Practice of Object Systems, John Wiley & Sons, USA , 1997, pp. 253–263.
[13] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis, "Advanced clone-analysis to support object oriented system refactoring", Proceedings of the Seventh Working Conference on Reverse Engineering, IEEE Computer Society, USA, 2000, pp. 98-107.
[14] J. Pérez, "Overview of Refactoring Discovering Problem", ECOOP 2006, Doctoral Symposium and PhD Students Workshop, Nantes, France, 2006.
[15] P. Bottoni, F. Parisi-Presicce and G. Taentzer "Coordinated distributed diagram transformation for software evolution", Electronic Notes in Theoretical Computer Science, Elsevier B.V, 2002, pp. 59-70.
[16] N. Van Eetvelde and D. Janssens, "A hierarchical program representation for refactoring", Electronic Notes in Theoretical Computer Science, Elsevier B.V., 2003, pp. 91-104.
[17] Available at http://submain.com/products/codeit.right.aspx, on 15.02.2009.
[18] F. Simon, F. Steinbruckner and C. Lewerentz, "Metrics based refactoring", Proc. European Conf. Software Maintenance and Reengineering, IEEE Computer Society, USA, 2001, pp. 30–38.
[19] I. G. Czibula and G. Serban, "Improving Systems Design using a Clustering Approach", IJCSNS International Journal of Computer Science and Network Security 6, 2006, pp.40-49.
[20] P. Berkhin, "A Survey of Clustering Data Mining Techniques", Springer, 2006, pp. 25-71.
[21] J. Zhang, Y. Lin and J. Gray, "Generic and Domain-Specific Model Refactoring using a Model

Transformation Engine", Springer, 2005, pp. 199-217.
[22] T. Massoni, R. Gheyi and P. Borba, "Formal Refactroing for UML Class Diagrams", 19th Brazilian Symposium on Software Engineering (SBES), Brazil, 2005, pp. 152-167.
[23] O. Ben Hadj Alaya, W. Charfi, M. Romdhani and M. Maddeh, "Metamodel refactoring Library of primitives of transformations", INSAT, Tunisia, 2008.
[24] D. Campbell and M. Miller, "Designing RefactoringTools for Developers", Second ACM Workshop on Refactoring Tools, Nashville, Tennessee, 2008.
[25] Refactoring tools, available at http://www.aivosto.com/vbtips/refactoring.html, on 15.02.2009.
[26] C.Stevenson and A. Pols, "An agile approach to a legacy system", Extreme programming and agile processes in software engineering, Springer, 2004, pp.123-129.
[27] Jrefactory, available at http://jrefactory.sourceforge.net/, on 20.02.2009.
[28] K. Maruyama and K. Tokoda, "Security-Aware Refactoring Alerting its Impact on Code Vulnerabilities", Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific, Beijing, 2008, pp. 445-452.