Legacy Issues in Industrial Software Development

Johan Kraft and Joel Huselius
Mälardalen Real-Time Research Centre
Mälardalen University
{johan.kraft, joel.huselius}@mdh.se

This report presents the results of the graduate course ”Legacy Issues in Industrial Software Development”, which was organized at Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University during winter 2006-2007. The course was open for all graduate students at the department, of which five actively participated in the course. PROGRESS is a major research project at MRTC, as well as a strategic research centre at Mälardalen University, aiming at enabling predictable component-based development of embedded systems. PROGRESS spans several areas, organized in clusters: component-based development, life-cycle processes, platform technology, dependability, and legacy systems.
The PROGRESS legacy cluster focuses on legacy issues of industrial software. There are however two common views of the term “legacy software”. In the traditional view, legacy software is no longer maintained but used as there no suitable alternative, for various reasons. In this view, the user of the software has the legacy problem, not the developer (they typically don’t support the software anymore). In our view, legacy software is software that is still maintained and further developed, but is becoming increasingly hard to maintain due to its long history of changes. The size and complexity of the system have increased over many years. Such systems often contain millions of lines of code, and due to personnel turnover many of the original developers and architects are no longer available. This view focuses on the problems of developing the software; the user of the software does not experience any explicit legacy problem.
The PROGRESS legacy cluster has two major goals. The first goal is to find a method for how to analyze complex embedded legacy systems with respect to quality attributes like performance (end-to-end response time) and resource usage. The main issue here is how to extract an analyzable model from the existing legacy system. The second goal of the legacy cluster is to enable integration of existing legacy code in embedded systems developed using component-based technology. Closely related work is very limited. In research areas such as component-based software engineering (CBSE) and real-time systems many results exists regarding how systems can be constructed assuming that one builds a new system starting from zero, but how to handle legacy systems is not considered. There are many results related to legacy systems in the reverse engineering community, but typically focusing on documentation/comprehension purposes rather than enabling analysis of behavioural properties. Moreover, the special requirements of embedded real-time systems are most cases not considered. The PROGRESS legacy cluster intends to bridge this gap.

Motivation

The course was intended as a bootstrapping activity for the PROGRESS legacy cluster and a framework for presenting and discussing existing scientific results related to the PROGRESS legacy cluster. To get credits for the course, the participants should give a lecture regarding a relevant topic and write a paper on another relevant topic. The papers were not expected to report own novel research results, but rather intended to present and relate exiting results relevant to the legacy cluster from an area selected by the participant. The papers were peer-reviewed and presented at a final course workshop. Five graduate students in different phases

of their education participated in the course, which resulted in five lectures and five papers. As there were only five participants, all papers were peer-reviewed by all four other participants. Revised versions of the papers are collected in this report, where any review comments have been addressed.
The lack of closely related research has naturally influenced the course contents. Many results discussed in this course will not have a direct impact on the future work of the legacy cluster, but rather serve to motivate the research and to help understanding the "big picture" with respect to the problems of legacy software and the state of related research areas.

Deliverables

In the first part of the course, the following lectures where given by the course participants:

- Lecture 1: *Course Introduction and Legacy Systems in General*, given by Johan Kraft, December 6th, 2006.
- Lecture 2: *Re-engineering/Migrating Legacy Code to new technologies*, given by Joel Huselius, December 13th, 2006:
- Lecture 3: *Reverse engineering and program comprehension*, given by Hongyu Pei-Breivold, January 24th, 2007.
- Lecture 4: *The E-Cares research project*, given by Farhang Nemati, January 24th, 2007.
- Lecture 5: *Specification, Verification of Real-Time Operation System and IDE Framework Design*, given by Yue Lu, Feburary 28th, 2007.

The slides to these presentations are available at the course webpage[1].

The second part of the course resulted in the following papers, which was presented at the final course workshop on April 4th, 2007:

- [1] - Farhang Nemati, *Using Reengineering Techniques for the Component Based Software Systems*
  As Component Based Software Engineering (CBSE) becomes more frequently used, the need for reengineering methods for CBSE is accentuated. The paper presents an overview, where results in this field are related. Requirements for reengineering component based real-time systems are provided.

- [2] - Hongyu Pei Breivold, *Quality Impact Analysis in Refactoring*
  Evolution deteriorates software quality. Refactoring, i.e., restructuring internals without effecting externals by small steps of transformations, can serve to tackle this deterioration. The paper relates a general process of refactoring, and details the various steps and constraints that are involved in the process. A comparison of three methods of refactoring is presented.

- [3] - Joel Huselius, *On the Difficulties of Maintaining Legacy Systems: Can it be solved by Model-Based Development?*
  The academia is less interested in maintenance than motivated by the costs of maintenance paid by industry. With the recent academic focus on model based development (MBD), maintenance issues can be directed, but since MBD is not legio in legacy systems, methods are needed to refactor code oriented legacy systems into

[1] http://www.idt.mdh.se/kurser/legacy/, May 2007.

model based systems.

- [4] - Johan Kraft, *Software Aging and the Code Decay Phenomenon*
  Code decay describes the increased complexity and reduced quality in software subjected to suboptimal maintenance. The paper relates a definition of code decay, provides motivation to how and why code decay arises, and finally discusses the measuring of code decay.

- [5] - Yue Lu, *An Introduction to Renaissance method: A Method to Support Legacy System Evolution*
  The paper provides an introduction to the Renaissance method, which is supporting reengineering primarily for business software, and has been developed in cooperation with industry. In addition, the paper also relates fundamental considerations and strategies of reengineering.

Notably, all papers are related. Three papers describe aspects of concrete activities in industrial software development, i.e. methods to apply when further developing legacy systems: reengineering [1], refactoring [2] and maintenance [3]. The paper about software aging [4] describes the forces that motivate refactoring and reengineering activities in a legacy context, while [5] describe strategies for managing legacy systems, when to apply e.g. refactoring. The relations between the papers are depicted in
Figure 1.

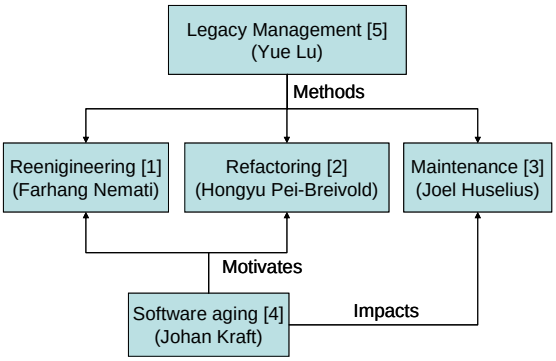


Figure 1: Relations between papers

Conclusions

With this report, we present the findings of the course “Legacy Issues in Industrial Software Development”. The lectures and the papers produced during the course served to highlight important lessons in the area of software evolution, and as a pre-study for our continued research. It is interesting to note that even though the participants were free to choose paper topics, the resulting papers are related, all have some overlap to at least one other paper. This is a very nice result, which indicates that all papers are relevant for the course focus, legacy systems. The paper presents and explains several terms of which some are not well defined in literature. We have identified several related works, in different areas, not previously known in the group. The seminars organized during the course generated many interesting discussions, which served to communicate our view of different issues among the participants. This document summarizes the course and provides a base for new course instances, where this report can be included as course material. We conclude that the desired goals of this course, to serve as a bootstrapping activity and as a framework for discussing related work, have been achieved.

Papers – Table of Contents

Using Reengineering Techniques for the Component Based Software Systems

Farhang Nemati

Department of Computer Science and Electronics
Mälardalen University, Västerås, Sweden
farhang.nemati@mdh.se

Abstract

Many reengineering techniques have been presented in the software engineering communities and some of them have been successful in the evolution and maintenance of complex legacy systems, but so far the target of these techniques has mostly been traditional software such as modular software. On the other hand Component Based Software engineering (CBSE) is growing in both academic research communities and industry, because the costs of the software development, maintenance, and evolution can be decreased by the benefits that CBSE offers, e.g. reusing components, better communication protocols, etc. As CBSE grows, the importance of using experiences of the reengineering community to migrate the traditional software to component based software and also to mature the component based development is revealed. This paper is an overview of using reengineering techniques in CBSE. We present a summary of component based technology in general, and in real-time context in particular. Then we review how CBSE can benefit from reengineering techniques. We also describe the extra considerations that should be taken into account when using the reengineering techniques for real-time components.

1. Introduction

Reengineering of complex big systems is important because often huge investments have been done on them, therefore they can't be thrown away. Often a legacy system is maintained during many years and during the lifecycle of the system many changes are applied to it that make the system even more complex. The target of reengineering a legacy system can be maintenance, evolution or performance improvement of the system. So far the reengineering has mostly dealt with the systems of non component based software such as modular software. As component based development grows in the software engineering community, the need for using the reengineering techniques in this area is revealed. According to [1], recently it has been shown that reengineering techniques can be useful in the context of the new technologies like component based software. Although these techniques are in their early stages, some successful developments have already been done by using the techniques. Reengineering techniques can be used to provide a migration framework toward the component based software engineering (CBSE) and also to mature evolution of the component based software products that have been developed recently [1]. During lifecycle of component based legacy systems, ad-hoc changes make the system more complex and harder to understand; they decrease cohesion between components and increase coupling among them, on the other hand often these changes are not well documented or reflected in the existing models [3]. Also according to [3] component recovery, by using reengineering

techniques, is used to retrieve components from the existing systems. Component based software engineering has recently been considered by real-time systems community as well; therefore using component based reengineering techniques in this area is also a challenge.

2. Component Based Software Engineering (CBSE)

The base of CBSE is the concept of component. There are many definition of what a component is; according to [4] a component is a reusable unit of deployment and composition which is closely related to an object and therefore, in some aspects, component based development is an extension of object-oriented development. However components differ from objects in many concepts such as granularity, deployment, composition and the process of development, etc.

Besides components there are some other concepts such as interface, contract, framework and pattern that are related to CBSE; these concepts in [4] are defined as following:

- An *interface* specifies the access points to a component, and thus helps clients to understand the functionality and usage of a component.
- A *framework* describes a large unit of design, and defines the relationships within a certain group of participants. According to [6] a component framework provides a variety of runtime services to support and enforce a component model.
- *Contracts* provide component specification; they focus on specification of conditions in which a component interacts with its environment.
- *Patterns* define recurring solutions to recurring problems on a higher abstract level, and in this way they enable the reuse of the solutions.

Figure 1 depicts a component pattern in which the relations between concepts in CBSE are illustrated.

According to [5], CBSE is mainly used:

- To provide support for the development of systems as assemblies of components
- To support the development of components as reusable entities
- To facilitate the maintenance and upgrading of systems by customizing and replacing their components.

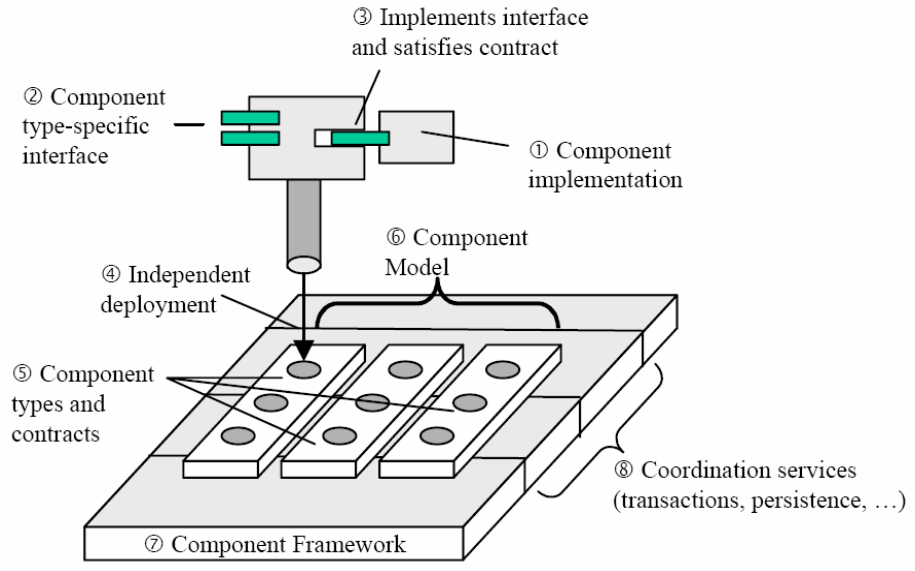


Figure 1: Relations between Concepts of CBSE [6]

3. Real Time Components

In real time systems besides expected logical results of the system, timing constraints play a big role and they have to be satisfied. One of main requirements in a real time system is the predictability of the timing attributes of the system, e.g. maximum response time of a task, deadlines, etc. [7]. According to [7] developing real time components is harder and more complex than none real time component, first because a real time component should satisfy timing constraints, second the resources in the embedded real time context are limited and finally real time applications should often run for a long period without errors and need for debugging. Requirements expected from real time components are divided into communication, synchronization and timing attributes [7].

4. Reengineering Techniques for Component Based Software

In [1] two reengineering techniques for component based software are considered:

1. Reengineering to support the maturation and evolution of component based software

In this case it is considered that a valuable amount of the component based software is available. Reverse engineering is used to extract component based concepts. Feasibility and accuracy of reverse engineering transformations very much depend on the component model and the information is concerned to be extracted. According to [1], recovering information of internal structure of components is usually possible, but recovering connections can be easy if connections are externalized, i.e. CCM, or can be complex if the connections are too much buried in the code, i.e. EJB or COM. Due to the mechanisms used, e.g. polymorphism, the connections can not be recovered by only using source code analysis, but also by using the behavior of the system during run time, therefore both static analysis (including source code , configuration files, etc) and dynamic analysis (by recording components behavior during run time) techniques of reengineering should be used to extract the architecture of the components and their connections.

Most of the reengineering techniques can be adjusted for component based software if the information extraction regarding components and connections is provided. The Reengineering techniques can be used for the evolution of the language of component model as well as for the evolution of the software; some parts of the software may be specified with previous versions of the component model language that can be replaced with the new versions.

2. Reengineering to support the migration to component based technology and integration with traditional ones

Using reengineering techniques for recovering components and connections out of a component based software, as described in the previous section is only appropriate for the software products that have used component based development. Reengineering techniques

can also be used to transform traditional software, e.g. modular software, to component based software by reusing the traditional software parts and wrapping it into components. This also depends on how good the target component technology can wrap the code written in traditional software, but this is not enough and more effort is needed to adapt the legacy code into component based technology since component based technology provides better communication protocols such as event triggered communication and also some component based mechanisms, e.g. EJB provide other services like transaction management and asynchronous messaging [1].

4.1. Establish Cooperation between Distributed Component Based Software

In [2] a component based reengineering methodology is proposed to provide cooperation between components in applications that are composed of distributed components. These components are not synchronized and are not aware of exchanging information. In this top-down approach, establishing cooperation between components starts from the highest level (constitution of workgroups) to the lowest level (the components that compose the workgroups). A workgroup is a set of eventually distant components with a common goal [2]. An optimal management of components needs a good organization of workgroups; this improves the performance regarding the amount of processed information and timing attributes which is suitable with a real-time context. The approach provides a cooperative platform that automatically manages the distribution of components involved in the cooperation. The solution deals with reusing of existing components and proposes developing of a layer above them to provide cooperation between them. The solution proposed by [2] consists of six steps:

1. **Workgroup level:** In this step workgroups in the application are identified; the identification of the workgroups is done according to constraints such as geographical constraints, method of work, etc. These constraints often correspond to the structural divisions of the firm [2]. The elements of cooperation are events, messages and the exchanged data. For each workgroup these elements are identified and the information they need and the information they produce is described with a natural language into a dictionary. For each of the cooperation elements an entry is added to the dictionary: (name, type, semantic description).

2. **Component level:** The existing components are identified in this step. Then it's identified of what workgroups the components can be members; the information in the dictionary (about the role of each component) is used to do this identification. The cooperation elements (events, messages, output and input data) related to each component are also identified in this step. The structure of the workgroups is dynamic which means that all member components may not be present at the same time. Also depending on the state of the application, a component can be a member of more than one workgroup; in this case the sharing of components is organized according to application constraints.

3. **Dynamic group management:** So far the elements of the cooperation have been identified, not the cooperation itself, meaning that the addressees (workgroups or components) of events, messages and produced data are not identified. In this step the constraints for each component, enter/exit and to/from workgroups, are revealed; this is done by dynamic rules. Dynamic rules are ECA rules which means when an event E

occurs, if the condition C is verified, the action A is executed [2]. Several workgroups can share a component, in this case for each of the workgroups the component has a dynamic rule associated to the workgroup. These rules show when a component has to enter into or exit from a workgroup, the rules will consider the global state of the application when evaluating a condition and the action triggered by the rule modifies the global state and therefore a new distribution of data should be done [2].

4. **Links and creation of missing cooperation elements:** In this step the focus will be on linking the elements that are defined in the dictionary during previous steps. A link may adjust the format of information to be compatible to its addressee. Some of the cooperation elements may not be linked to any entities within the application since they may be inputs (are not produced by any entities of the application), or outputs (are carried to outside of the application). Based on the dictionary, links to elements are established. First inter group links, .e.g. between components of a workgroup, then between workgroups are defined. Sometimes needed information is produced by composing the information in the dictionary, meaning that some of the links can't be explicitly extracted from the dictionary. If it's not possible to produce some needed information (neither explicitly nor by composing existing information), some elements must have not been discovered, if it's so we have to go back and explore the missed elements. If the exact information is not at all producible, another information (maybe less precise) can be identified. For composition and format the information to be compatible to its addressee some rules are used that are called detective rules. A specification language is used for definitions, links and creation of cooperation elements [2].

5. **Formal Check:** Before implementing the results they should be verified formally if all cooperation elements can effectively be produced. The specification language provides the possibility to formally check the results.

6. **Derivation the Specification Language into Rules:** To implement the obtained results, the specification language is derived into ECA and detective rules. The rules then will be integrated into the platform to manage the distribution of information. The rules also will manage the dynamic aspect of the workgroups, e.g. sharing components.

5. Reengineering of Real-Time Components

Reengineering of real-time components is more complex than of not real time components, because in all phases of reengineering (reverse engineering, restructuring and forward engineering) not only functional aspects of the system but also timing attributes of the system should be taken into consideration. Especially for a safety critical system, in which a failure may lead to a catastrophic accident, the timing constraints should be satisfied, e.g. deadlines shouldn't be missed. In reengineering of a real time system composing of components, besides guarantying the behavior of each component, composition of the components should also guarantee communication, synchronization and timing attributes of the whole system; this problem is called composition problem [7]. Most of reengineering techniques that have been used, especially the techniques that are used for reengineering of traditional real time systems can be extended, either to migrate to component based real time software or for evolution of the existing real time component based software. In [8] a dynamic verification of safety critical real time software for reusing components is proposed, in this approach they present a framework for determining when the components in a safety critical real time

system can be reused, when they should be retested, and when only some parts of the system needs retesting. However the reliability of components, no matter if they are extracted form traditional legacy system or are reused, should be achieved before implementing them into the system. In [8], by examples that have led to catastrophic accidents, they show that the reliability of components is not necessarily preserved when reusing them.

Lüders in [9] presents an evolutionary approach in which adoption of existing component models for embedded real-time systems is explored. As benefits of this adoption he refers to using existing development environments, reusing or adopting existing components for the real-time domain, and simplifying integration with applications from other domains.

6. Summary

In this paper we presented an overview of using reengineering techniques for component based software engineering. First we pointed out the importance of reengineering and the extension of reengineering techniques to CBSE. We described what CBSE is, and what a component means in general and in real time in particular. We described how techniques in the reengineering community can help the maturation and evolution of component based software. We also described how these techniques can be used to migrate to component based technology and integrate them with traditional software. Then we pointed out a six step approach that provides cooperation between components in an application composed of distributed components. Finally we explained what should be considered in the reengineering components from a real time perspective.

References

[1] J.M. Favre, H.Cervantes, R. Sanlaville, F.Duclos, and J.Estublier. Issues in Reengineering the Architecture of Evolving Component-Based Software. In proceeding of *Working Conference on Reverse Engineering (WCRE'2001),* 2001.

[2] P. Roose. ELKAR: a component based re-engineering methodology to providecooperation. In proceeding of *25th Annual International Computer Software and Applications Conference (COMPSAC 2001), 2001.*

[3] R.Koschke. Atomic architectural component recovery for program understanding and Evolution. In proceeding of *International Conference on Software Maintenance, 2002.*

[4] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Basic Concepts in CBSE. *Chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, to appear, 2002.*

[5] G. T. Heineman, and W. T. Councill. Component Based Software Engineering: Putting the
Pieces Together, Reading, MA: Addison-Wesley, 2001.

[6] F. Bachman, et all. Technical Concepts of Component-Based Software Engineering. Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carengie Mellon University, 2001.

[7] D. Isovic, and C. Norström. Components in Real-Time Systems. *Chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, to appear, 2002.*

[8] H. Thane, and A. Wall. Testing Reusable Software Components in Safety-Critical Real-Time Systems. *Chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, to appear, 2002*

[9] F. Lüders. An Evolutionary Approach to Software Components in Embedded real-Time Systems. PhD thesis, Mälardalen University, Sweden, 2006.

Quality Impact Analysis in Refactoring

Hongyu Pei Breivold

hongyu.pei-breivold@mdh.se

Abstract

Software evolution is an important issue in software engineering. During the evolution of software intensive systems, reducing system complexity and improving software quality are the most after striving goals. Refactoring technique is regarded as one of many efficient ways to achieve the goals. However, to make appropriate refactoring decisions is a challenging task and demands quality impact analysis as main inputs. Consequently, the ability to estimate and measure the refactoring impact on quality characteristics of the system has become critical. Various techniques have emerged and they assess quality impact either qualitatively or quantitatively in terms of specific quality metrics. They differ from each other in terms of principles, concepts and analysis capabilities as well. All these factors lead to the difficulties in selecting appropriate refactoring impact analysis technique, thus influencing the widespread application of quality impact analysis techniques.

In this paper, we will present essential requirements for quality impact analysis techniques in refactoring and apply these requirements as criteria to review three techniques that are used for assessing the effect of refactoring on quality characteristics.

We believe that the requirements addressed in this paper can serve as a base and checking metrics for software analysts in selecting appropriate quality impact analysis technique that suits their specific needs in refactoring. They can also act as a starting point as well for further improvement and development of quality impact analysis techniques in refactoring validation process.

1. Introduction

In the past few years, we have witnessed an enormous expansion in the use of software in business, research, industry, and critical infrastructure systems such as civil, telecommunications and medical systems. While new software and systems are developed at tremendous speed, they become legacy systems at fast speed too. Several factors contribute to this phenomenon.

(1) Software aging

Software ages quickly because the overall business requirements change at tremendous speed. Therefore, in order to survive the competition and maintain a leading position among competitors, the software systems need to keep up to the changing demands from the customers to meet new functionality requirements and to overcome any existing system limitations. The inevitable aging of all software systems have turned even rather new object oriented systems into legacy systems [6]. This factor corresponds well with Lehman's Laws of software evolution regarding continuing change, i.e. an E-type program that is used must be continually adapted, and else they become progressively less satisfactory [8].

(2) Increasing software complexity

Most of the software intensive systems nowadays become more and more complex due to the constantly incoming new requirements and evolution of technologies. It is quite common that as

an evolving program is continually changed, its structure deteriorates [12], especially when we have too tight schedule to consider and analyze the consistency of design and have to fix problems before deadline and within budget. Consequently, complexity increases unless work is done to maintain or reduce it [9].

(3) Declining software quality

Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to decline as it is evolved [9]. The extreme time-to-market pressure contributes to the degradation of software in the sense that quick fixes are done in the code without considering the potential impact of code change to the program structure and software architecture.

The knowledge and experience of software developers can influence a lot to the outcome of software system evolution as well. Developers need to have deep understanding and knowledge about software architecture, quality attributes, developing software-intensive systems and modern software engineering techniques. Misuses of object-oriented principles such as encapsulation, inheritance, etc. also contribute to the declining software quality.

Meanwhile, poor documentation leads to lack of understanding of the software, resulting in declined software quality.

To summarize, as the software is enhanced, modified and adapted to meet new requirements, the system becomes more complex. Therefore, while designing and implementing a large scale and complex system has been a challenging task, evolving and maintaining the software system to reduce complexity and meanwhile meet quality attribute requirements during the system life cycle has become even more challenging.

Today, a lot of attention and effort have been focused on how to cope with and reduce software complexity and increase software quality. One emerging technique is called refactoring. It is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. Its heart is a series of small behaviour preserving transformations. Each transformation which is also called a 'refactoring' does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it is less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring [4].

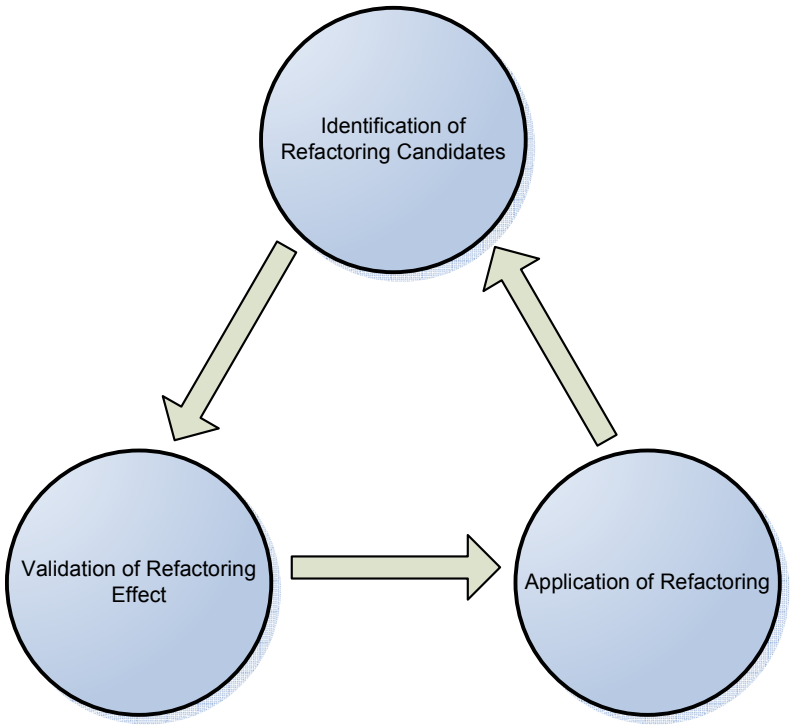
Refactoring is regarded in general as one of the important ways to improve software quality. Therefore, the ability to measure and estimate qualitatively and/or quantitatively the impact of refactoring on quality characteristics is of great interest because it helps in identifying and selecting refactoring strategies and decisions in order to apply appropriate refactoring to meet quality requirements. In this paper, we will present the essential requirements for quality impact analysis in refactoring and use these requirements as criteria to compare several techniques assessing the effect of refactoring on quality characteristics.

The rest of the paper is structured as follows. Section 2 describes the role of quality impact analysis in refactoring. Section 3 describes the essential requirements for quality impact analysis in refactoring. Section 4 presents different quality impact analysis techniques in refactoring. Section 5 gives a comparison of the presented analysis techniques against the requirements. Section 6 concludes the paper.

2. The role of quality impact analysis in refactoring

Refactoring is defined as 'process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure' [5]. The goals of refactoring are to extract a reusable component, to improve consistency among components [11], to improve software design, to make software easier to understand and etc. The overall goal is to improve quality characteristics, such as maintainability, of the software [1].

Refactoring is an iterative process that involves three main phases, i.e. identification, validation and application [7] as shown in Figur 1.



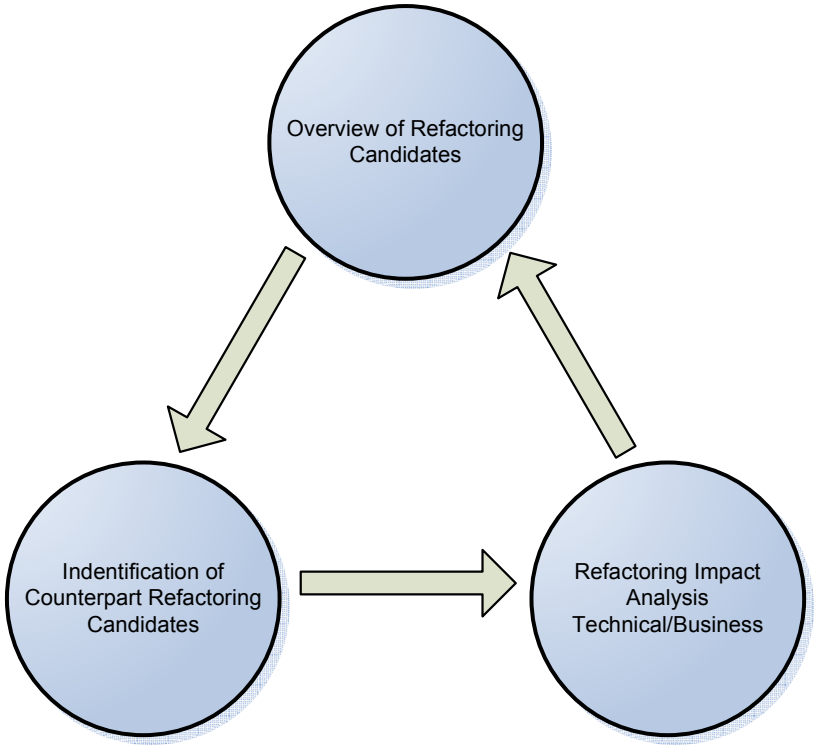
Figur 1 Refactoring Process

The identification phase identifies refactoring candidates, i.e. where the software should be refactored. The validation phase validates the refactoring effect. It provides inputs to which refactoring should be applied to the identified candidates through assessing the effect of refactoring from both technical and business perspectives. Technical assessment estimates the effect of refactoring on quality characteristics such as complexity, maintainability of the software. Business assessment estimates the cost and effort on applying refactoring. The application phase applies the identified refactorings.

The underlying assumptions throughout the refactoring process are that the applied refactoring preserves behaviour and that the consistency between refactored artefacts and other software artefacts in the system can be guaranteed, in the sense that requirement specification, architectural design documentation, software code and test specification, etc. should match with each other.

In this paper, we focus on quality impact analysis in validation phase. The validation phase of refactoring effect as depicted in Figur 2. It is also an iterative process consisting of analyzing the overview of refactoring candidates, identifying the counterpart refactoring candidates and executing refactoring effect analysis from technical and business perspectives, i.e. quality impact analysis with cost/effort analysis in consideration.

As stated earlier, the validation of refactoring effect provides invaluable inputs to making refactoring strategies and decisions. An ideal situation is to be able to validate the quality impact of a certain refactoring and estimate the cost and effort for the refactoring before applying it in the real system. Therefore, quality impact analysis plays a central role in refactoring process since it provides a primary basis for software evolution. Consequently, it is important that techniques for estimating the impact of refactoring on the quality characteristics of the software can take into account the essential requirements that will be described in section 3.



Figur 2 Validation Phase of Refactoring Impact

3. Requirements for quality impact analysis in refactoring

This section describes the essential requirements that techniques for quality impact analysis in refactoring should support. These requirements can serve as checking metrics for choosing appropriate quality impact analysis technique in refactoring. They provide an indication as well on how the quality impact analysis techniques in refactoring can be further improved and developed. The identified requirements are described in the following sessions.

3.1 Analysis requirements

Quality impact assessment consists of both technical analysis and business analysis. Technical analysis focuses on the refactoring impact on quality characteristics, such as maintainability. Business analysis focuses on cost and effort in applying refactoring, including the impact analysis on synchronizing requirement specification, design documentation, software code and test specification for each refactoring.

- Quality concerns
  In order to help with the identification and prioritization of counterpart candidates, the analysis process should include identification of important quality attributes and

refinement of the selected quality attributes into concrete and specific quality requirement expressions. Any trade-offs among quality attributes need also to be identified.

- Cost and effort concerns
  In order to help with making appropriate refactoring decision and prioritization of counterpart candidates from business perspective, the analysis process should include cost and effort analysis, because it is critical to have the ability to identify refactoring candidates that have great refactoring effect potential and reasonable cost and effort, especially when there is extreme time-to-market pressure. Therefore, cost and effort analysis provides assistance in making long term refactoring plans and sometimes constrains the achievement of certain quality characteristics as well.

  Cost and effort analysis is the change impact of a refactoring, ranging from applying refactoring on the code level to synchronizing relevant changes in documentation related to the refactoring, such as requirement specification, architectural design documentation, test specification, etc.

- Architectural concerns
  Software architecture plays a central role in software systems. It has tight connection to the system's quality requirements in the sense that software architecture constrains the quality attributes [3]. Software architecture is expressed in terms of various architectural styles, design patterns and design models that contribute to the achievement of certain quality characteristics [16]. On the other hand, refactorings improve the internal structure of the software without altering the external behaviour of the system. Therefore, the quality impact analysis techniques need to support identifying, extracting, expressing and managing architectural styles and design patterns.

- Analysis level
  Quality impact analysis can be applied on various levels, such as method level, class level and architectural design level.

3.2 Assessment requirements

There are different types of quality assessment: qualitative or quantitative assessment. Both complement each other. Qualitative assessment can serve as a base and indication for further quantitative assessment if necessary. On the other hand, quantitative assessment can further prove the correctness of qualitative assessment through measurement values.

- Qualitative assessment
  Although qualitative assessment can not be graphed or displayed in terms of mathematical expressions, it is still an important indication in supporting the refactoring decision and strategy making process.

- Quantitative assessment
  A quantification metrics need to be selected for measurement in the case of a quantitative assessment. The measurement values provided from the quantitative assessment are of great importance and can act as evidences in comparisons, especially when trade-off concerns are involved and decisions need to be made among various alternatives during quality impact analysis process.

- Trade-off concerns in quality analysis
  Three aspects exist regarding trade-off concerns:

Firstly, counterpart refactoring candidates can not be applied at the same time [7].

Secondly, quality impact analysis from both technical and business perspectives can sometimes come into conflict with each other in the sense that a comprehensive refactoring needs to be made from technical point of view while the cost and effort analysis from business perspective constrains the application of the refactoring due to the time-to-market pressure and limited budget.

Thirdly, trade-offs among quality attributes need to be considered.

In all three cases, a comprehensive plan for refactoring and trade-off decisions should be decided before any further step in refactoring process.

3.3 Tool requirements

Quality impact analysis is a repetitive and complicated process. Consequently, it is important to have good tool support in assisting effective execution of the quality impact analysis process.

4. Quality impact analysis techniques

Many different techniques can be used to measure or estimate the impact of a refactoring on quality attribute [10]. We present three representatives in this paper. They differ from each other in terms of concepts, principles and analysis capabilities.

4.1 Coupling metrics

Kataoka et al. propose coupling metrics as a quantitative evaluation method to measure the maintainability enhancement effect of a program refactoring [7].

There are various aspects of maintainability of a program, such as coupling, cohesion, size and complexity, description, etc. All of them contribute to the maintainability quality attribute of the software system in the sense that low coupling and high cohesion modules enhance system maintainability, simple and small modules are easy to maintain, appropriate naming rules help with program understanding.

Among the above mentioned aspects, coupling is selected as a maintainability quantification metrics and classified into three categories, i.e. return value coupling, parameter coupling and shared variable coupling. Coefficients for respective coupling need to be defined in order to represent the impact degree of respective coupling in a method refactoring. Thereafter, measurement of the metrics is made before and after the refactoring and comparison becomes explicit, with the delta value indicating the maintainability enhancement effect.

An example is shown in Figur 3. It is results from applying refactoring methods that are supposed to reduce coupling among methods, such as Extract Method and Extract Class. The column ‘before’ shows the coupling metrics value of the target method while the column ‘after’ shows the coupling metrics value after refactoring the target method and the effect of refactoring in parentheses.

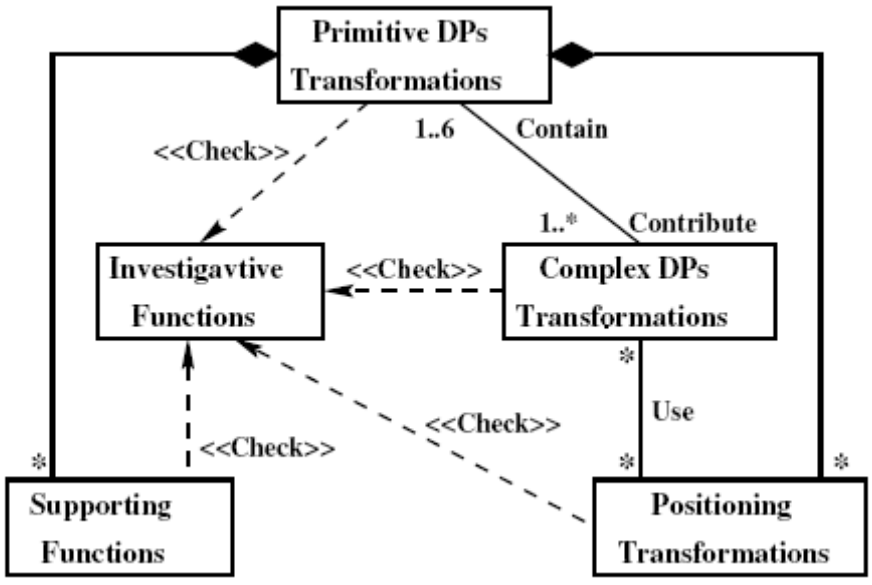
case	sbj.	before	after(effect)	average(effect)
1	*B*	10.4	2.8(7.6)	3.6(6.8)
2	*C*	12.1	9.0(3.1)	8.3(3.8)
3	*B*	25.2	9.0(16.2)	9.0(16.2)
4	*B*	26.4	1.7(24.7)	13.8(12.6)
5	*A*	126.0	26.0(100.0)	28.3(97.7)

Figur 3 Example of refactoring impact analysis result [7]

The application of this evaluation method demands extensive experiences in making good judgement on finding appropriate coefficient values to calculate the coupling metrics value.

4.2 Transformation using soft-goal graph

Tahvildari and Kontogiannis propose a re-engineering transformation framework for object oriented legacy systems. This transformation framework correlates non-functional requirements with design patterns to guide transformation process [14]. The definition and refinement of quality requirement is based on NFR framework [2], where a soft goal interdependency graph is used to support modelling of design rationale. The transformation process is based on a transformation meta-model illustrated in figur 4.



Figur 4 Meta-model of the transformation [14]

The transformation process defines concretely step by step on how to implement a transformation:

Step 1: Evaluate if pre-conditions for applying a transformation hold, using investigative functions.

Step 2: A step by step implementation is defined, using supportive functions and positioning transformation.

Step 3: Evaluate if specific conditions hold after a transformation is applied, using investigative functions.

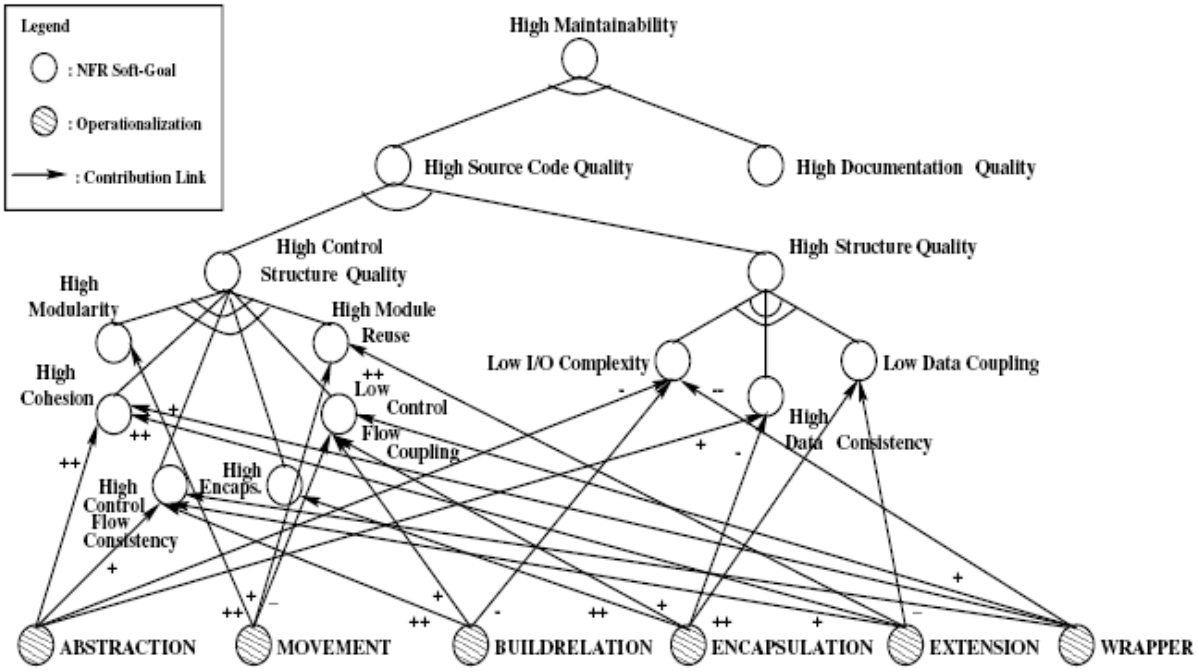
Because the use of design patterns has impact on system quality attributes, the goal of the transformation framework is to associate each design pattern transformation to one or more soft goals.

The transformation framework has the feasibility to analyze any quality attributes of a software system though Tahvildari and Kontogiannis focused on maintainability as an illustrating example. The soft goals that are refined from maintainability are:

(1) Coupling: low control flow coupling and low data coupling enhance system maintainability.
(2) Cohesion: high cohesion modules are easy to maintain.
(3) Modularity: programs that have many direct interrelationships between any two random parts of the program code are less modular than programs where those relationships occur mainly at well-defined interfaces between modules [19]. High modularity enhances system maintainability.
(4) Encapsulation: a software module hides information by encapsulating the information into a module that are most likely to change, thus protecting other parts of the program from changing [18] and enhancing system maintainability.
(5) Complexity: low I/O complexity modules are easy to maintain.
(6) Consistency: high control flow consistency and high data consistency enhance system maintainability.
(7) Reuse: high module reuse enhances system maintainability.

Six categories of primitive design pattern transformations are identified, such as Abstraction, Extension, Movement, Encapsulation, etc. and they can be combined to produce complex design pattern transformations related to various design patterns.

A qualitative association (positive or negative impact) is identified between each design pattern transformation and the above mentioned soft goals, i.e. the aspects of maintainability. Figur 5 shows the association between primitive design patterns and maintainability soft goal graph. The association between complex design patterns and maintainability soft goal graph can be further derived.



Figur 5 Qualitative association between primitive design patterns and maintainability soft goal graph [14]

4.3 Metric-based transformation

Tahvildari and Kontogiannis propose also another framework for detection and correction of design defects on class level in object oriented legacy systems [15].

A catalogue of object-oriented metrics is used as indicators for automatically detecting where a particular transformation can be applied to improve the software quality. The object oriented metrics are classified into three categories: complexity metrics, coupling metrics and cohesion metrics. Examples of these object oriented metrics are CDE (Class Definition Entropy), NOM (Number of Methods), WMC (Weighted Methods per Class) in complexity category, DAC (Data Abstraction Coupling) and RFC (Response For a Class) in coupling category, LCOM (Lack of Cohesion in Methods) and TCC (Tight Class Cohesion) in cohesion category.

The detection process in the framework includes checking design principles and detecting violations by using different design heuristics [13], such as key classes, one class – one concept. The correction process in the framework is based on analyzing the impact of meta-pattern transformations, extracted from the previous approach (transformation using soft goal graph in session 4.2), on these object oriented metrics. The impact of meta-pattern transformations on these object oriented metrics is shown in Figur 6.

When classes that need refactoring are detected by applying certain design heuristics, suitable meta-pattern transformation can be selected based on Figur 6 to solve the violation problem to design heuristics, thus to enhance system maintainability.

Meta-Pattern Name	Description	CDE	DAC	LCOM	NOM	RFC	TCC	WMC
ABSTRACTION	adds an interface to a class which enables another class to take a more abstract view of the first class by accessing via interface	−	*NI*	*NI*	+	+	−	*NI*
EXTENSION	constructs an abstract class from an existing class and creates an extends relationship between the two classes	*NI*	+	+	*NI*	+	+	−
MOVEMENT	moves parts of an existing class to a component class and sets up a delegation relationship from the existing class to its component	−	+	+	*NI*	−	+	−
ENCAPSULATION	weakens the association between two classes by packaging the object creation statements into dedicated methods	−	+	+	*NI*	−	+	+
BUILDRELATION	operates the relationship between the classes in a more abstract fashion via an interface	+	+	*NI*	*NI*	−	+	+
WRAPPER	wraps an existing receiver class with another class in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps and similarly any results of such requests are passed back by the wrapper	+	+	*NI*	*NI*	+	+	−

Figur 6 Impact of meta-pattern transformations on object oriented metrics [15]

This approach combines using metrics for quality estimation and performing transformation based on soft goal graphs. It analyzes the interaction between software transformations and metrics.

5. Comparison of quality impact analysis techniques

A review of the presented techniques for quality impact analysis in refactoring is made against the essential requirements described in section 3, which form the base for the comparison criteria. These

three techniques differ from each other in terms of concepts, principles and analysis capabilities. Therefore, we believe it is suitable to analyze them as representatives against the essential requirements that we have extracted in section 3.

5.1 Analysis requirements

– Quality concerns

Coupling metrics technique focuses on maintainability and analyzes only coupling aspect. Therefore, there is no trade-off consideration in terms of various quality attributes.

Soft goal graph transformation focuses on various quality attributes and identifies refinement of the quality attributes into concrete and specific quality requirement expressions through soft goal graph. Trade-offs among quality attributes can be detected through analyzing the soft goal graphs.

Metric based transformation focuses on quality attributes in terms of a set of object oriented metrics, which can be used to assess system qualities.

– Cost and effort concerns

None of the three techniques takes into account cost and effort concerns as constraints to quality impact analysis in refactoring.

– Architectural concerns

Coupling metrics technique focuses on method level. Therefore, there is no strong emphasis on high level architecture.

Soft goal graph transformation technique takes into account the primitive and complex design patterns and analyzes quality impact from architectural design level.

Metric based transformation technique detects refactoring candidates through applying design heuristics and analyzes the correlation between design patterns and object oriented metrics.

– Analysis level

Coupling metrics approach is on method level.

Metrics based transformation is on class level.

Transformation using soft-goal graph is on architectural design level.

5.2 Assessment requirements

– Qualitative assessment

Both soft goal graph transformation technique and metric based transformation technique use qualitative assessment.

– Quantitative assessment

Coupling metrics technique uses quantitative assessment.

– Trade-off concerns in quality analysis

There are no trade-off concerns in terms of quality attributes in coupling metrics technique.

Both soft goal graph transformation technique and metric based transformation technique take into account the quality trade-off factors by refining the quality attribute in soft goal

interdependency graph and analyzing the positive and negative impact of design patterns on quality attribute refinements.

The trade-off concerns between technical assessment and business assessment are not applicable because none of the techniques take into account cost and effort analysis.

5.3 Tool requirements

Coupling metrics technique uses Refactoring Assistant as tool support for bad-smell detection. Based on this detection result in terms of coupling metrics and interviews with software developers, the most serious problems in terms of maintainability are extracted for further analysis.

NFR Assistant [17] can be used in the soft goal interdependency analysis graph for both soft goal graph transformation technique and metrics based transformation technique.

Generally, tool support in quality impact analysis is desired to facilitate effective execution of impact assessment of refactoring.

A summary of the comparisons of the quality impact analysis techniques against the essential requirements is shown in Figur 7.

Requirements		**Coupling Metrics**	**Soft Goal Graph Transformation**	**Metric Based Transformation**
Analysis	Quality	Coupling aspect only	yes	yes
	Cost & Effort	no	no	no
	Architectural	no	yes	yes
	Analysis Level	method	architectural design	class
Assessment	Qualitative	no	yes	yes
	Quantitative	yes	no	no
	Trade-off	no	partial	partial
Tool Support		no	no	no

Figur 7 Summary of quality impact analysis techniques comparison

6. Conclusion

In order to reduce system complexity and enhance software quality, refactoring technique has emerged as one of the many efficient solutions. However, refactoring process is complicated and covers a set of comprehensive activities. Quality impact analysis is one of them and plays a central role in assisting refactoring decision making during the refactoring process as well as in making contributions to successful system and software evolution.

This paper addressed the essential requirements for quality impact analysis techniques in refactoring. We reviewed three representative quality impact analysis approaches that differ from each other in terms of concepts, principles and analysis capabilities. Based on the analysis of these techniques, we investigated how they match against the essential requirements.

We believe that the requirements that are derived from this paper can serve as a base and checking metrics for software analysts in selecting appropriate quality impact analysis technique that suits their

specific needs. They can also act as a starting point and indication for further improvement and development of quality impact analysis techniques in refactoring process.

References

[1] BD. Bois and T. Mens: Describing the Impact of Refactoring on Internal Program Quality. (2003)
[2] L. K. Chung, B. A. Nixon, E. Yu and J. Mylopoulos: Non-Functional Requirements in Software Engineering. (2000)
[3] I. Crnkovic and M. Larsson: Building Reliable Component-Based Software Systems. (2002)
[4] M. Fowler: http://www.refactoring.com (visited 2007)
[5] M. Fowler: Refactoring Improving the Design of Existing Code. Addison-Wesley (1999)
[6] A. Hessellund: Refactoring as a Technique for the Reengineering of Legacy Systems. (2004)
[7] Y. Kataoka et al: A Quantitative Evaluation of Maintainability Enhancement by Refactoring. (2002)
[8] MM. Lehman: Laws of Software Evolution Revisited (1996)
[9] MM. Lehman and JF. Ramil: Rules and Tools for Software Evolution Planning and Management. (2001)
[10] T. Mens and T. Tourrwé: A Survey of Software Refactoring. (2004)
[11] WF. Opdyke: Refactoring Object-Oriented Frameworks. (1992)
[12] S. L. Pfleeger: The nature of system change. IEEE Software (1998)
[13] A. J. Riel: Object-Oriented Design Heuristics. (1996)
[14] L. Tahvildari and K. Kontogiannis: A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph. (2002)
[15] L. Tahvildari and K. Kontogiannis: A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations. (2003)
[16] L. Tahvildari and K. Kontogiannis: On the Role of Design Patterns in Quality-Driven Re-Engineering (2002)
[17] Q. Tran and L. Chung: NFR – Assistant: Tool Support for Achieving Quality (1999)
[18] Information Hiding. http://en.wikipedia.org (visited 2007)
[19] Modularity in Computer Science. http://en.wikipedia.org (visited 2007)

On the Difficulties of Maintaining Legacy Systems: Can it be solved by Model-Based Development?

Joel Huselius (joel.huselius@mdh.se)

May 22, 2007

Abstract

One of our observations is that maintenance is a much neglected subject in contemporary research. We motivate the need for maintenance in industry and discuss difficulties, different research results, and their potential.

1 Maintenance

In the context of this report, a *legacy system* has all or some of the following properties: it consists of millions of lines of code, it is maintained by a large team of engineers from several generations,[1] it contains code originated several years ago, and it is expected to function for many more years to come. Real examples of these systems can easily be found within many domains such as automation, automotive, and telecom industries.

Maintenance is defined by the IEEE [9] as:

> *"...modifying a software system or component after delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment."*

There are four types of maintenance:

1. Corrective: repair of discovered faults
2. Adaptive: environmental adaptations (e.g. upgraded hardware)
3. Perfective: functional enhancements due to new and/or revised requirements
4. Preventive: changes to increase maintainability of software

[1] Several generations, i.e. the set of engineers that have contributed to the system is a superset of the set of engineers currently working on the system.

Industries around the world are continuously maintaining their respective legacy systems. From the industry point of view, this is a major issue (and has been for some time [1, 15, 22]). However, from the academia, little attention has been given to these matters [1, 16]. For example, according to their web page, the IEEE organized 759 conferences in 2005. Six (6) out of these had a clear focus on maintenance.[2] To put this in perspective, twenty nine (29) of the conferences had the word "Wireless" in the title. The lack of research in this field may be due to that it is difficult to objectively classify software maintenance as anything more detailed than arbitrary changes to arbitrary things [6].

1.1 Motivating maintenance

In his seventh *law of software evolution* [13, 14], where an *E-type program* is a program that solves a problem or implements a computer application in the real world, Lehman proclaims that:

> *"E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to changing operational environment."*

Thus, software maintenance is needed to keep the quality of the legacy system. In an uncontrolled quality decline the legacy system is made obsolete prematurely, which leads to even larger costs than required for maintenance.

Apart from legacy system quality, also the legacy system complexity is jeopardized by poor maintenance. Lehman's second law (still concerning E-type systems) states that:

> *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*

According to intuition, a less complex legacy system is less difficult to maintain, and a more complex legacy system is more difficult to maintain (this has also been shown in [4]). Thus, maintenance should be a continuous effort throughout the life cycle of the legacy system, and affordable efforts aimed to maintain the complexity within acceptable bounds will limit the cost of maintenance.

1.2 A process for maintenance

Maintenance to direct a given change request consists of a set of sub-activities (slightly adapted from [16]):

1. Understanding the change request
2. Understanding the system and its structure (i.e. reverse engineering [3])

[2] (1.) The Working Conference on Reverse Engineering, (2.) the Annual Reliability and Maintainability Symposium - Product Quality & Integrity, (3.) the 9th European Conference on Software Maintenance and Reengineering, (4.) the IEEE International Conference on Software Maintenance, (5.) the IEEE Workshop on Source Code Analysis and Manipulation, and (6.) the IEEE 13th Workshop on Program Comprehension.

3. Designing the implementation of the change request
4. Implementing the change request
5. Determining ripple effects
6. Designing new change requests based on ripple effects
7. Performing maintenance for the new change requests
8. Testing that the system fulfills all previous and new requirements

1.3 Requirements on maintenance

Short-term requirements on maintenance includes avoiding to violate previously stated requirements and choosing a simple change implementation with limited ripple effects. A more intricate long-term requirement is to maintain the scalability of the legacy system so that future maintenance can be easily performed. This may require choosing another alternative than *the* simplest change implementation, which makes the act of choosing a design implementation slightly more complex. Another important requirement is of course to reduce the cost of maintenance.

1.4 Example of tool assisted maintenance: MORALE

Abowd et al. [1] present the tool supported development framework Mission ORiented Architectural Legacy Evolution (MORALE). The framework is mission oriented as opposed to being oriented by technical criteria, i.e. a proposed change is seen in the context of the goals and behavior of the legacy system. Abowd et al. state that architectural modifications are those that are most difficult and most time consuming. In an attempt to reduce costs in the introduction of architectural modifications, MORALE allows risk calculation for failure to implement specified architectural changes and an analysis to determine what parts of the legacy system that is affected by an architectural change. As a pre-stage to these, MORALES also includes a manual reverse engineering process for extracting architectural information from a legacy system.

Though some tool support exists [18], the amount of manual labor implied is prohibitive.

2 Maintenance through model-based development

Models represent an abstraction from the legacy system. According to the IEEE [8], an *abstraction* is:

> *"A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information."*

Abstractions are made and used all the time, they are fundamental for efficient communication and cognition. In reality, it is often difficult to choose the most appropriate abstraction for a given situation. If the abstraction is too high, the value of the

information is reduced. If the abstraction is too low, processing the information is tedious. Finding the most suitable abstraction is a balance between reducing the amount of information and preserving potentially relevant information.

2.1 Model-based development

Model-based development (a.k.a. model-based engineering) aims to improve the efficiency and the quality of the development process (including maintenance) by making it more formal and more mechanical. As opposed to traditional code-oriented development, where the source code is the primary view of the developer, models are the primary view of the development and maintenance process [12]. They are used as media to formalize, convey, develop, and preserve the properties and requirements of the legacy system. After that the model is completed and validated, the intension is that the push of a button (or the technical equivalent) should generate the application code. The hope is also that the extensive use of models that describe the functional and temporal properties of the implementation will facilitate reuse, automated validation, automated verification, and model-based analysis (e.g. impact analysis).

In terms of maintenance, model-based development has the potential to ensure that old requirements are not violated, and the possibility to apply model-based analysis on designs of change implementations at an abstract level before full implementation could help to reduce costs and maintain the complexity of the legacy system.

2.2 Requirements on model-based development

In order to use models as the primary view throughout the life cycle of the legacy system and its parts, the models must present a uniform view to avoid divergence between models [19]. However, different activities in development and maintenance are focused on different aspects of the legacy system [20]. Modeling represents an abstraction for a given purpose, and different purposes have different requirements regarding what information is important and what can be abstracted. Thus, the modeling framework should allow for separation of concerns.

2.3 Maturity of model-based development

Research in model-based development focus on domain-specific modeling, meta-modeling for architectural descriptions, code generation from models, and analytic methods for supporting the development and maintenance of models [10]. Though much research has been conducted in these areas, it is probably safe to say that model-based development is not yet ready to make its mark in industry on a broad front.

Legacy systems maintained in industry are generally code-oriented, and as Littlejohn et al. [17] point out: wholesale redevelopment is cost prohibitive, and prior investments must be preserved. In order to take model-based development into industry, reverse engineering methods must be developed that either atomically transform code into models in a revolutionary manner, or find other means of making a more gradual or evolutionary shift of development paradigm. In code-oriented development, the existing code-base is a valuable asset of the company, it is the result of enormous

investments and has taken many man-years using other development paradigms to perfect. Any technique for introducing model-based development must do so with such detail and quality that the existing code-base can be discarded. This poses very high demands on the techniques for introducing model-based development. Contrary to intuition, it is likely that the more gradual shift towards model-based development is associated with a higher stake. Due to the accumulated work performed to make the shift, and the reduction in work efficiency during the shift, the stake of a failed paradigm shift is higher than in the case of the "big-bang". The risk however, is likely to be lower with a gradual approach that has the ability to adopt dynamically to issues that arise during the paradigm shift. Most sources seem to agree that a gradual paradigm shift is preferable [11, 17].

Regarding revolutionary means to introduce model-based development, there are several methods for automating reverse engineering, but we do not know of any method that allows the introduction of model-based development. We exemplify by mentioning two methods:

2.3.1 Machine learning of timed automata

Grinchtein et al. [5] present a method for reverse engineering to time deterministic event recording automata (i.e. transition guards of the automata are mutually exclusive) based on traces of the program. Their approach is to view this as a learning problem:

In *machine learning* [2] a *Learner* is concerned with hypothesizing a model from a system by asking a *Teacher* if the system accepts a given behavior trace (i.e. asking *membership queries*). Which membership queries to ask is given by inconsistencies in the Learner's currently available data; if a set of behavior traces that the Learner thinks are equal yields different answers when the Teacher is asked, an inconsistency exists. By reformulating the hypothesized model and asking a set of more detailed queries, these inconsistencies are resolved. When a Learner has obtained a sufficient confidence in the correctness of the hypothesis (i.e. when all inconsistencies has been resolved), the hypothesized model is checked with an *Oracle* to determine its correctness (i.e. asking *equivalence queries*). If the model is incorrect, the Oracle will present a counter example that can be used to extend the model. Implementation issues involve realizing the Teacher and the Oracle. Implementing a Teacher may be prevented due lack of *reproducibility* in the system [21]. Regarding complexity: Since membership queries need to be asked until all inconsistencies are resolved, the number of membership queries can potentially be large. The final model is not guaranteed to have the same structure as the system.

2.3.2 From code to model

Holzmann and Smith [7] introduced a method called *Modex* (acronym for MODel EXtractor) for reverse engineering from code. They describe static model extraction as a hierarchical process, which makes it very comprehensible. As a proof of concept, they present a one shot experiment on a large telephone application. Modex takes the source code of the implementation as input, which makes the model accurate to the implementation.

Apart from the source code of the implementation, four additional types of inputs are required; these can be seen as non-operational models that are supplied as input. It is claimed that the production of these inputs is a simple task, but that some updates to the input can take in the order of hours to complete.

3 Conclusions

In this paper, we have discussed the differences between the industry and academic views on maintenance. We have exemplified a framework for maintenance, and discussed the possibility of easing maintenance by introducing model-based development.

Our conclusion is that model-based development provides features and capabilities needed in industry to ease maintenance, but more research is needed to facilitate its introduction. The model-based technologies must mature, and introduction of model-based development must be directed.

References

[1] Gregory Abowd, Ashok Goel, Dean Jerding, Michael McCracken, Melody Moore, William Murdock, Colin Potts, Spencer Rugaber, and Linda Wills. MORALE. mission oriented architectural legacy evolution. In *Proceedings of International Conference on Software Maintenance*, pages 150–159, October 1997.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

[3] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[4] Virginia R. Gibson and James A. Senn. System structure and software maintenance. *Communications of the ACM*, 32(3):347–358, 1989.

[5] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. *Lecture Notes in Computer Science*, 4137:435–449, 2006. In Proceedings of the 17th International Conference on Concurrency Theory.

[6] David Hearnden, Paul Bailes, Michael Lawley, and Kerry Raymond. Automating software evolution. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 95–100, September 2004.

[7] Gerard Johan Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *Transactions on Software Engineering*, 28(4):364–377, April 2002.

[8] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, December 1990. IEEE Std. 610.12-1990.

[9] IEEE. *IEEE Standard for Software Maintenance*. IEEE, June 1998. IEEE Std. 1219-1998.

[10] IEEE Computer Society. *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, March 2006.

[11] Ivar Jacobson and Fredrik Lindström. Re-engineering of an old system to an object-oriented architecture. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pages 340–350, October 1992.

[12] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Theodore Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[13] Meir Manny Lehman. Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[14] Meir Manny Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, October 1996.

[15] Bennet P. Lientz, E. Burton Swanson, and Gerry Edward Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

[16] Mikael Lindvall, Seija Komi-Sirviö, Patricia Costa, and Carolyn Seaman. Embedded software maintenance: A dacs state-of-the-art report. Technical Report DACS SOAR 12, Data and Analysis Center for Software, January 2003.

[17] Kenneth Littlejohn, Michael V. DelPrincipe, and Jonathan D. Preston. Embedded information system re-engineering. *IEEE Aerospace and Electronic Systems Magazine*, 15(11):3–7, November 2000.

[18] Spencer Rugaber. A tool suite for evolving legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 33–39, August 1999.

[19] Bernhard Schtz. Model-based engineering of embedded control software. In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 53–62, March 2006.

[20] Tivadar Szemethy, Gabor Karsai, and Daniel Balasubramanian. Model transformations in the model-based development of real-time systems. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 177–186, March 2006.

[21] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[22] Stephen W. L. Yip, Tom Lam, and Stephen K. M. Chan. A software maintenance survey. In *Proceedings of the First Asia-Pacific Software Engineering Conference*, pages 70–79, December 1994.

Software Aging and the Code Decay Phenomenon

Johan Kraft
Mälardalen University
johan.kraft@mdh.se

Abstract
This paper describes the issue of software aging and the phenomenon known as code decay. The paper is gives an introduction to the subject, describes the driving forces behind code decay and also presents research on measurement of code decay.

Introduction

Software does not wear out in the conventional sense, like a physical object, but do experience an aging process. It is commonly recognised that software that has been maintained for some time and where many changes has been made, typically is harder to change than new software. The accumulated effect of previous maintenance operations is often described as software aging. The main symptom of software aging, and the main motivation for studying software aging, is the resulting increase in maintenance costs. The many changes made, often in a suboptimal manner, makes the code increasingly complex and hard to understand, both locally (e.g. complex control flow within a function) and on a higher, architectural level (e.g. many dependencies between modules). This phenomenon is commonly known as *code decay.*

A definition of code decay is found in [2], "*a unit of code (in most cases, a module) is decayed if it is harder to change than it should be, measured in terms of effort, interval and quality*". In this definition, the *effort* is the number of person-hours required for the change, *interval* is the calendar time required and *quality* is the absence of errors introduced when performing the change.

There exists another use of the term software aging; when the performance and/or reliability of software deteriorate during runtime, e.g. due to resource management issues. This is described in, e.g., [5]. The focus of this paper is however the aging of software due to the changes made between different revisions of the software.

For industrial software systems, the concept of software aging and code decay is of high importance. Industrial software systems are often very large, containing millions of lines of code, and therefore constitute a huge investment for a company – hundreds or thousands of person-years of development time. Such software systems can not easily be replaced as it is important to preserve this investment. As a company's profit from the development investment is proportional to the time it can be used in the company's products, it is very important that the software system can live as long as possible. Industrial software systems are therefore often maintained for many years, or even decades, after the initial release. Maintenance operations are inevitable and often frequent, due to the market demand for new features and the many errors typically discovered post-release.
The costs for software maintenance have been estimated to account for 50-80% of the total lifecycle cost of software in general [3]. For industrial software systems, that typically is very long-lived, this is most likely a very conservative estimation. The issue of software aging and code decay has therefore a large economic impact for developers of industrial software systems. Imagine a medium size software organisation, with 100 software developers that spend 75 % of their budget (development time) on maintaining existing code. If the

phenomenon of code decay is responsible for even a 5 % decrease in development efficiency, this corresponds to a cost of 4 person-years.

In general, the cost for inefficient software development is huge. According to a recent study [4] by the National Institute of Standards and Technology (NIST) at the U.S. Department of Commerce, software bugs cost the U.S. economy an estimated $59.5 billion annually. The study concluded that more than a third of these costs could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects, i.e. finding an increased percentage of errors closer to the development stages in which they are introduced. But how does this relate to software aging? Many bugs are probably the result of aging, decayed software. In order to improve development efficiency, increasing the awareness and prioritization of long-term software quality, i.e. maintainability, is at least as important as novel methods for software verification.

The purpose of this paper is to give an overview of software aging and code decay, why it occurs and how it can be measured.

Factors behind code decay

Aging is an issue for all successful software products, as they will be changed frequently during their lifetime. Only software that no one uses remains unchanged. Lehman has formulated several commonly known "laws" (observations) regarding the nature of software maintenance [1]. Three of the laws are of special relevance here:

The law of Continuing Change: "*An E-type program that is used must be continually adapted else it becomes progressively less satisfactory*".

The law of Increasing Complexity: *"As a program is evolved its complexity increases unless work is done to maintain or reduce it"*.

The law of Continuing Growth: "*Functional content of a program must be continually increased to maintain user satisfaction over its lifetime."*

An E-type solves a problem or implements a computer application in the real world. All industrial software products are in this category.
The Continuing Change can be explained by the fact that the environment that the program is used in will most likely change (e.g. technical standards, laws, business rules), which require adaptations of the software. Moreover, new features and/or better performance will be requested, either by the customers directly or in response to competing products, in order to keep market shares. Continuing Change is also due to the errors will be reported long after the initial release, which will require fixes to be made.
As observed by Lehman (Increasing Complexity), the changes made increases the complexity of the software. This partly due to the increasing functional content, and thereby increasing size of the implementation (Continuing Growth), but also due to code decay caused by the changes.

As noted by Lehman, the complexity increase can be compensated by efforts striving at improving the system's maintainability, i.e. *perfective maintenance*, also known as *refactoring*. However, since refactoring does not improve the functionality of the system, such activities are often not prioritized until the level of code decay has become very high. If time

is allocated for more frequent refactoring activities, the effort required for each refactoring activity will be much smaller and the system will generally have better maintainability. There are several contributing factors behind the code decay phenomenon.

- Software Engineering competence
- Personnel turnover
- Focus on short-term goals
- Not designing for change

Software Engineering competence One issue is the fact that many software developers are not software engineers [6]. They have other academic backgrounds, like mechanics, control theory, physics, mathematics etc. They know how to write code, but many lack professional education in large scale software development, e.g. sound design principles and quality assuring activities like reviews.

Personnel turnover Another issue is the personnel turnover during the system lifecycle. Many developers are inexperienced, at the company or in general. Moreover, a lot of the design rules used in system's original architects have not been explicitly documented and have been lost as experienced developers have left the company. Thus, many maintenance operations are performed by developers that don't know the original design rules of the system and therefore don't follow them when implementing changes. In order to fully understand the system after such changes, it is necessary to understand both the original design and the rationale behind the later changes. After many such changes, nobody fully understands the product [6].

Focus on short-term goals A big issue is the time pressure from management or customers on meeting short term goals. This makes developers take short-cuts, that violate design guidelines or in other ways constitute sub-optimal solutions. Time pressure also makes developers neglect updating the design documentation to reflect their changes. Focus is on getting the program to behave as requested, while long-term quality, i.e. maintainability and quality of design documentation, has low priority. In [6], the author suggests that managers not prioritize long-term software quality in order to maximize short term results, to improve their visible management results and thereby promotion chances. This is probably hard to avoid, as the long-term software quality is hard to measure. However, one must remember that short term goals are very important too, long-term quality is not relevant if the company goes out of business! A balance is necessary.

Not designing for change Programmers in general do not design for future changes, but focus on achieving desired functionality. This is related to the issued of programmer competence and pressure on short-term goals. As future changes are inevitable, designing for maintainability constitute a long-term investment in software quality. As put in [6], "*Designing for change is designing for success*".

The listed factors behind code decay are really symptoms of the immaturity of software development today. This is a serious and increasing problem, considering the high and rapidly growing use of software in products.

Measuring Code Decay

The measuring of code decay has been discussed in several works, typically case studies where source code metrics are collected from different revisions of large software systems and analyzed in order to identify modules with code decay problems. In this way, refactoring

efforts can be focused on the modules with greatest need. Code decay is always relative to previous system revisions due to the definition "… harder than it used to be", and can not be quantified in absolute terms. The known works that measure code decay focus on identifying the most decayed modules of a system. In [7], a classification scheme is proposed, dividing legacy components (modules) into three categories, green, yellow and red components.

- Green components are considered "normal" and do not have any significant code decay. Some fluctuations are of course possible, but changes can in general be made without problems.
- Yellow components are components where the code decay has exceeded a lower limit and where attentions is necessary in order to avoid future problems, if the code decay increase further. Such components are candidates for directed refactoring activities.
- Red components are components where the code decay have exceeded an upper limit and where maintenance is significantly harder than usual.

However, it is not trivial to determine where the two limits are (green-yellow and yellow-red), as it is dependent on many factors and hardly can be decided in absolute terms, but need to be subjectively selected based on relative measures. The authors present three classes of metrics used for analyzing the code decay; faults, structural changes and size changes.
The history of fault can be used to identifying the components that often have faults and thereby are likely to suffer from code decay. By comparing structural properties of the different modules between revisions, it is possible to identify major structural changes, which may lead to reduced maintainability. Size changes of the modules could also a measure, major changes in size could be a warning of code decay. The result from the case study performed in [7] is however weak, as they used too few data points, they only compared two revisions. In [8] the authors present a more extensive study, where 8 revisions have been compared. 28 different measures where collected of 130 software components. Using these measures, they where able to identify a large number of "healthy" components and a small number of "problematic" components, on which many changes have been necessary.
In [2] the authors present a set of code decay indices (CDI's) which they have used to investigate the existence of code decay in a very large telecom switching system, containing 100.000.000 lines of code. The analysis was performed on based on the entire change management history - 15 years of changes. They observed an increase over time in the number of files modified by a change and a decrease in modularity. Moreover, through statistical analysis they found several factors that seemed to influence the amount of faults of a module, including the frequency and recency of changes.

Conclusions

Code decay is a symptom of software aging and occurs due to the accumulated effect of all changes performed during the system's maintenance phase, which often lasts for many years. The symptom of code decay is treated by perfective maintenance, also known as refactoring. The reason behind the code decay is that many changes are often performed in a suboptimal manner due to many factors, e.g., time pressure, inexperienced personnel and poor design documentation. It is possible to measure code decay by comparing source code metrics between revisions, but it is always a relative measure, as code decay can not be quantified in absolute terms. Some academic results exist in this area, but the industry awareness of this issue is in general low.

References

[1] – M. M. Lehman, "Laws of Software Evolution Revisited", in *Proceedings of the European Workshop on Software Process Technology*, pages 108-124, 1996.

[2] – S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus, “Does code decay? Assessing the Evidence from Change Management Data”, *IEEE Trans. on Softw. Eng.*, v. 27, n. 1, pp. 1 – 12, 2001.
[3] – D. Gefen and S. L. Schneberger, “The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications”, in *Proceedings of the International Conference on Software Maintenance*, pages: 134 – 141, 1996.
[4] – “The Economic Impacts of Inadequate Infrastructure for Software Testing”, *Planning Report 02-3, Prepared by RTI for the U.S. National Institute of Standards and Technology*, 2002.
[5] – K. C. Gross, V. Bhardwaj, R. Bickford, “Proactive Detection of Software Aging Mechanisms in Performance Critical Computers”, in *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, 2002.
[6] – D. L. Parnas, “Software aging”, in *Proceedings of the 16th International Conference on Software Engineering*, Pages: 279 – 287, 1994.
[7] – M. C. Ohlsson and C. Wohlin, “Identification of Green, Yellow and Red Legacy Components”, in *Proceedings of the International Conference on Software Maintenance*, pages: 6-15, 1998.
[8] – M. C. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, “Code Decay Analysis of Legacy Software through Successive Releases”, in *Proceedings of the IEEE Aerospace Conference*, March 1999.

An Introduction to Renaissance method: A method to Support Legacy System Evolution

Yue Lu
Mälardalen University, Västerås, Sweden
yue.lu@mdh.se

Abstract

Legacy systems are often business-critical and are associated with high maintenance costs [2]. The *Renaissance* project is conducted in the aiming to manage the process of regaining control over the legacy systems. Recovering a stable basis using reengineering at first, and subsequently continuously improving the system by a stream of incremental changes[2], support the system evolution in *Renaissance*. It is pointed out that the system evolution is determined by three main factors, e.g. technical, business, and organisational [2]. *Renaissance* defines a process framework, a predefined number of evolution strategies, and information repository, and a generic set of personnel responsibilities. The method can be tailored to the needs of particular projects and organisations, instantiated to offer a cost / risk trade-off, and it is not prescriptive of particular tools and techniques. The *Renaissance* method is quite interesting. However, due to the lack of possibility of assessing the real system in our research filed, it is impossible to research whether it can work and show how much it can perform or should be improved in the legacy system that we are aiming at. Whereby, only the method is introduced in this paper with content from the references.

1. Legacy systems

Due to the time and efforts required to develop a complex system, e.g. which consists of millions of lines of codes, large computer based systems usually have a long lifetime. After being developed, the systems will progressively become less useful, if they can't be changed to response to the changing environments. Whereby, the needs for accommodating systems changes according to the new requirements from technical, business, and organizational perspectives, are important and necessary. A legacy system is an old system that remains in operation within an organisation [2]. Its compositions can be shown by the following figure 1 [1].

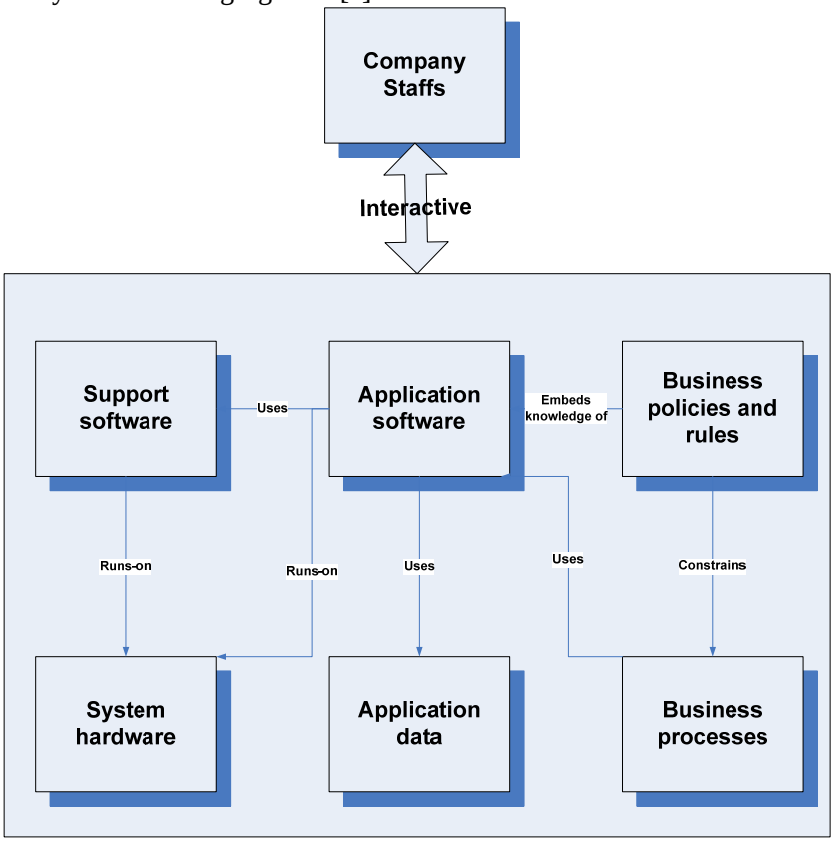


Figure 1 Legacy system components

Many of the legacy systems are business-critical, and hard, ineffective, expensive to be maintained due to the reasons for short anticipated system lifetimes, lack of well-organized system design documentations and experienced personnel, decreased maintainability caused by satisfying other design constraints, obsolete technology used and the immaturity of software engineering practice at the time of system construction etc.

Studies have shown that reengineering, where appropriate, is generally more cost effective and less risky than developing a new system (replacing) [4]. The *Renaissance* method [5,6] which is proposed in the *RENAISSANCE* project, an ESPRIT funded research project into software reengineering and software evolution, aims to support a controlled approach to system evolution.

2. Renaissance Method

2.1 System evolution method requirements

In the paper [2], there are four key requirements of a method to supports system evolution are identified, and can be shown by the Table 1[2].

R1.	The method should support incremental evolution.
R2.	Where appropriate, the method should emphasise reengineering, rather than system replacement.
R3.	The method should prevent the legacy phenomena from reoccurring.
R4.	It should be possible to customise the method to particular organisations and projects.

Table 1 Method requirements

2.2 Renaissance approach

In the *Renaissance*, they proposed a two-stage process for transforming legacy systems to evolvable systems. In the first stage, the reengineering, or even in some extreme cases, replacing, is used to recover a stable basis of system transformation to evolvable system. The costs and risks associated with the system evolution can be predicted after the system transformation is done. In stage 2, the further evolution of the evolvable system in its life cycle will be conducted continuously. The figure 2 [2] shows the *Renaissance* approach.

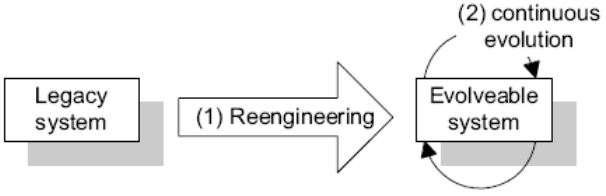


Figure 2 *Renaissance* approach

The *Renaissance* method comprises a classification of evolution strategies, a process framework, an information repository, and a set of responsibilities to be met in a typical evolution project. Each of these elements can be tailored to fit particular project and organisational factors [2].

2.3 Evolution strategies

There are six proposed evolution strategies in paper [2], which can be shown refer to the Table 2 [2].

Continued Maintenance	The accommodation of change in a system, without radical change to its structure, after it has been delivered and deployed.
Revamp	The transformation of a system by modifying or replacing its user interfaces. The internal workings of the system remain intact, but the system appears to have changed to the user.
Restructure	The transformation of a system's internal structure without changing any external interfaces.
Rearchitecture	The transformation of a system by migrating it to a different technological architecture.
Redesign with Reuse	The transformation of a system by redeveloping it utilising some legacy system components.
Replace	Total replacement of a system.

Table 2 Evolution strategies

Generally speaking, these options are not exclusive, and can be used in the same application. Since a system is composed of several programs, thus different options may be applied to different parts of the system.

2.4 Process Framework

The process framework proposed in the *Renaissance* is split into four high-level phases with identified key activities. The framework starts with Plan Evolution, which addresses the systems' long-term future, and involves assessing the current system from technical, business, and organisational perspective with a view to develop an evolution strategy[2]. Process framework can be shown with the combination of following figure 4 and Table 2 [2].

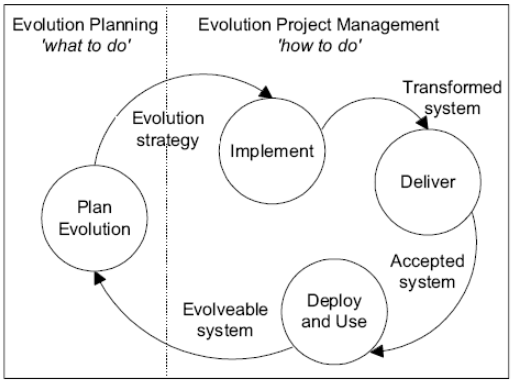


Figure 3 Abstract process model

Phase	Activities
Plan Evolution	Calibrate method Assess system Develop evolution strategy
Implement	Plan evolution project Design, transform and test system Prepare environment
Deliver	Migrate data Install system Train operators
Deploy and Use	Changeover system Evaluate system Document environment changes

Table 3 Key activities in *Renaissance*

The key effective evolution planning is thorough assessment of the legacy system [7]. The Assessing a system by using an attribute-rating scheme [1] means calculating measures for the system's technical quality and business value, and taking account of any organisational factors which might affect an evolution project [2].

From a business perspective, the key point is to decide whether the business really needs the system. From a technical perspective, the qualities of the application software, the system's support software, hardware are taken into account. Then the business value and system quality will be combined, to inform the decision on what to do with the legacy system[1]. To illustrate, it is proposed that an organization has 8 legacy systems. The quality and the business value of each of these systems is assessed and compared with others by plotting it on a chart showing relative business value and system quality [1].

This is illustrated in Figure 4 [1], where there are four clusters of systems.

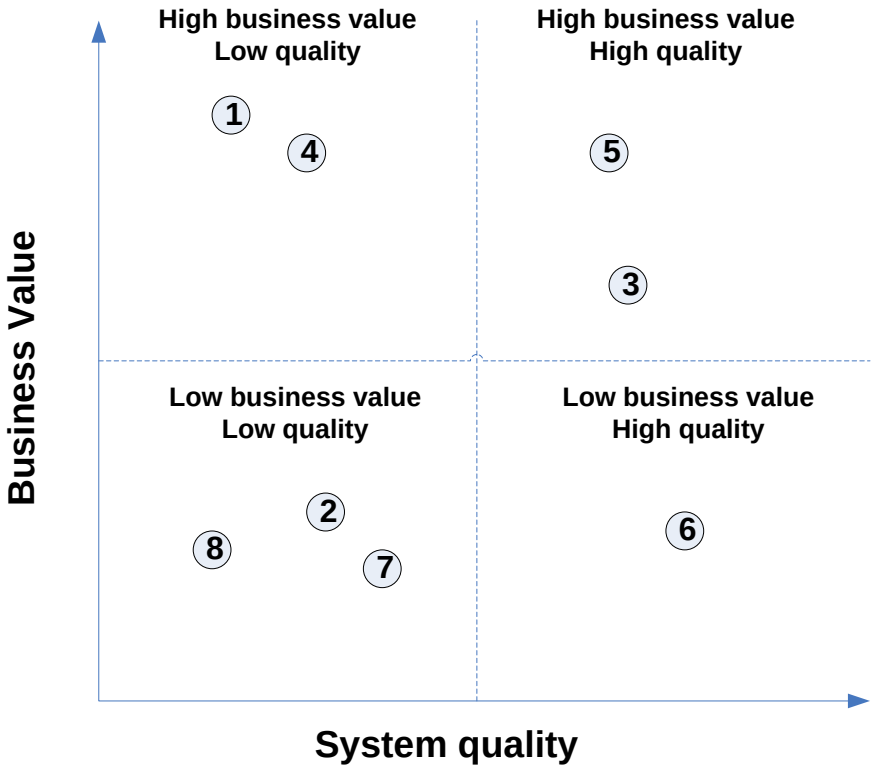


Figure 4 Legacy system assessments

To assess a software system from a technical perspective, both the system application and the environment in which the system operates, are taken into consideration. The environment consists of the hardware and all associated support software, e.g. compilers linkers, which are required to maintain the system. The changes in environment result in system changes, such as upgrades to the hardware or operating system. If possible, in the process of environmental assessment, measurements of system and its maintenance processes should be made. Example of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software [1].

Factors to be considered during the environment assessment are supplier stability, failure rate, age, performance, support requirements, maintenance costs, interoperability etc. Notice that these are not all technical characteristics of the environment. The reliability of the suppliers of the hardware and support software should be taken into account as well. If these suppliers are no longer in business, there may not be maintenance support for their systems.

To assess the technical quality of an application system, a range of factors are to be assessed, e.g. understandability, documentation, data, performance, programming, language, configuration management, test data, personnel skills etc, that are primarily related to the system dependability, the difficulties of maintaining the system and the system documentation. It will be helpful to judge the quality of the system if quantitative system data are collected. Although this data is often useful, collecting it can be very expensive and therefore impractical. Furthermore, there are no absolute values that may be used. The age and size of the system have to be taken into account when making quality judgments based on measurements.

2.5 Method Evaluation

During the *Renaissance* project, industrial partners were involved in evaluating the method. Each partner used applications with different technical, business, and organisational properties. Based on their findings, the method was refined. *Renaissance* method is fit for the system evolution method requirements described in the section 2.1 However, evaluators found that the method was better suited to managing medium-to-large projects than small projects, due to the overhead introduced in the project initial stage [2].

References

[1] Software Engineering, SOMMERVILLE, Chapter 2, Legacy systems, Page 39-41, Legacy system evolution, Page 504-509.

[2] Renaissance: A Method to Support Software System Evolution. Ian warren and Jane Ransom, Computing Department, Lancaster University.

[3] Lehman, M. M. and Belady, L. Program Evolution: Processes of Software Change. London: Academic Press. 1985.

[4] Ulrich, W. M. The Evolutionary Growth of Software Reengineering and the Decade Ahead. American Programmer, 3(10). 1990.

[5] Warren, I. (ed.) The Renaissance of Legacy Systems. Practitioner series, Springer. 2000.

[6] The Renaissance Method. RENAISSANCE project deliverable, D4.2. 1998. Jane Ransom, Computing Department, Lancaster University. Lancaster, LA1 4YR.

[7] Ransom, J., Sommerville, I., and Warren, I. A Method for Assessing Legacy Systems for Evolution. Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR). 1998.