Evaluating Software Evolvability

Hongyu Pei Breivold
ABB Corporate Research
721 78 Västerås, Sweden
+46 21 323243
hongyu.pei-breivold@se.abb.com

Ivica Crnkovic
Mälardalen University
721 23 Västerås, Sweden
+46 21 103183
ivica.crnkovic@mdh.se

Peter Eriksson
ABB AB
721 78 Västerås, Sweden
+46 21 344310
peter.j.eriksson@se.abb.com

ABSTRACT
Software evolution is characterized by inevitable changes of software and increasing software complexities, which in turn may lead to huge cost unless rigorously taking into account change accommodations. This has intensified the need on evolvable software systems that can correspond to changes in a cost-effective way. Nevertheless, although software evolvability is one of the most important quality attributes of software, it is not precisely defined today. Besides, the lack of evolvability model hinders us from analyzing, evaluating and comparing software systems in terms of evolvability. To address these issues, we distinguish software evolvability from maintainability in this paper and outline a suggestion for an evolvability model which analyzes software evolvability from various perspectives, as well as an evolvability evaluation method. The model and the method are evaluated through its application in an industrial automation system. The contribution of this paper is the initial establishment of an explicit definition of software evolvability, an evolvability model and an evolvability evaluation method that can be applied for large complex software-intensive systems.

Keywords
Software evolvability, maintainability, quality model

1. INTRODUCTION
Software maintenance and evolution are characterised by their huge cost and slow speed of implementation [3]. The ability to change and evolve software quickly and reliably has become a challenging issue for both software engineering community and industry.

Industry rarely develops new products from scratch [11]. New features, constraints and enhancements of most new products are usually built on top of the earlier versions of software products. This is due to the fact that in most cases, the cost of evolving software is lower than developing from scratch [20]. Typical examples are industrial automation systems. Since industrial automation systems are often long-lived software-intensive systems that can have a lifetime of 20-30 years, they are subject to changes and may undergo a substantial amount of modifications in order to be responsive to the constantly changing demands from the marketplace, stakeholders, business requirements, environment or technologies during their lifecycles. This implies that these software-intensive systems become more and more complex and may contain up to several million lines of code as the software is enhanced, modified and adapted during the software evolution process. Complexity increases unless work is done to maintain or reduce it [15]. These phenomena in continuing change and increasing complexity were recognized by Lehman and expressed in his well-known laws of software evolution [15]. The properties of large software systems noted by F. P. Brooks [6], e.g. software complexity, inevitable changes of software systems and invisibility in terms of software structure representation, further confirm the software evolution characteristics and exhibit the intensified need on evolvable software systems that can be long-lived and correspond to changes in a cost-effective way.

One way to ensure that any software system does not deteriorate as it is evolved is to provide feedback to the development team about the evolvability, since there is usually a potentially huge risk that the software systems will degrade and cost huge amount of money. Statistics have shown that the largest part of lifecycle costs for long-lived software systems is concerned with the evolution of the software [2] to cope with the challenges of the continuing change, increasing complexity and the tendency of declining software quality. Therefore, the systems' capability to cost-effectively adapt to and accommodate various changes has become essential for companies to survive in the competition and maintain a leading position among competitors. The inability to effectively and reliably evolve software systems means loss of business opportunities [3]. Consequently, there is strong demand to carry out software evolution efficiently and reliably, thus, to prolong the productive life of a software system.

Today, software needs to be changed on a constant basis with major enhancements within short timescale, in order to launch new products and services and keep up with new business opportunities, through coping with the changing environments and the radically changing requirements. All these put critical demands on the software system's capability of rapid modification and enhancement. In this sense, software evolution is one term that can express the software changes during software system's lifecycle and software evolvability is an attribute that describes the software system's capability to accommodate these changes with the condition of having the lifecycle costs under control. As software evolution activities are performed, essential characteristic software evolvability must be considered. Nevertheless, although software evolvability is one of the most important quality attributes or characteristics of software, it is not precisely defined today. It is not explicitly defined in any well-known quality models that we have investigated, e.g. McCall's quality model, ISO/IEC 9126, etc. Because of the lack of a standard definition, many people use software evolvability as synonymous to software maintainability. Although both have similarities in many senses, software maintainability and evolvability have specific focus, which has resulted in confusion in understanding and applying similar concepts designated

differently. Furthermore, software evolvability is affected by many factors and it is difficult to quantify.

Thus, in this paper, we intend to (i) show differences between software maintainability and evolvability, (ii) define a software evolvability model, (iii) identify the required sub-characteristics of software evolvability based on the analyses of several well-known quality models and comparisons between evolvability and maintainability, and (iv) evolvability evaluation method. This evolvability model is established as a first step towards quantifying evolvability, a base and check points for evolvability analysis and evaluation as well as evolvability improvement. Further we demonstrate the model and the method through an industrial case study.

The rest of the paper is structured as follows. Section 2 analyzes several existing well-known quality models, compares evolvability with maintainability and gives a definition of software evolvability and proposes the evolvability model. Section 3 presents evaluation of software evolvability using the model and relates it to different architecture evaluation methods that may be adapted for evolvability evaluation. A comparison between the evolvability model and the related methods is also addressed in this section. Section 4 presents a case study in applying the evolvability model and evaluation method. Section 5 concludes the paper and outlines the future work.

2. SOFTWARE EVOLVABILITY MODEL

To be able to define the evolvability model we start with a short analysis of different quality models in which we can find the elements of evolvability. In particular we analyze sub-characteristics of maintainability and defined the sub-characteristics of evolvability. Based on this analysis we provide the evolvability model.

2.1 Analysis of Quality Models

A quality model provides a framework for quality assessment. It aims at describing complex quality criteria through breaking them down into concrete sub-characteristics. The best known quality models include McCall [17], Boehm [4], FURPS [18], ISO 9126 [13] and Dromey [10]. The quality characteristics that are addressed in these quality models are summarized in Table 1. As shown in Table 1, although several quality attributes are correlated to software evolvability, e.g. adaptability, extensibility and maintainability, the term evolvability is not explicitly addressed in either of the quality models. On the other hand, this table provides useful inputs for the establishment of the software evolvability model, e.g. the identification of sub-characteristics of evolvability.

2.2 Evolvability

We define software evolvability as follows:

Definition: *Software evolvability is the ability of a software system to adapt in response to changes in its environment, requirements and technologies that may have impact on the software system in terms of software structural and/or functional enhancements, while still taking the architectural integrity into consideration.*

Software evolvability is both a business issue as well as a technical issue, since the stimuli of changes can come from both perspectives, including change of business models and business objectives, changes in environment, quality requirements, functional requirements, underlying technologies as well as emerging technologies.

This definition may remind of the definition of adaptive maintainability but there are principle differences, and differences in some characteristics, as discussed below.

Since maintainability is covered in most of the well-known quality models and it is generally considered as most related to evolvability, we will study the definitions of maintainability in order to make the definition and features of evolvability distinguishable. A summary of the definitions of maintainability in various quality models is presented in Table 2.

Table 1 Quality characteristics addressed in quality models

Quality Characteristics	McCall	Boehm	FURPS	ISO 9126	Dromey
Adaptability			x Supportability	x Portability	
Compatibility			x Supportability		
Correctness	x				
Efficiency	x	x		x	x
Extensibility			x Supportability		
Flexibility	x				
Human Engineering		x			
Integrity	x				
Interoperability	x			x Functionality	
Maintainability	x	x	x Supportability	x	x
Modifiability		x		x Maintainability	
Performance			x		
Portability	x	x		x	x
Reliability	x	x	x	x	x
Reusability	x				x
Supportability			x		
Testability	x	x		x Maintainability	
Understand-ability		x		x Usability	
Usability	x		x	x	x

Table 2 Definitions of maintainability in quality models

Quality Models	Maintainability Definition	Focus
McCall	The effort required to locate and fix a fault in the program within its operating environment	Corrective maintenance
Boehm	It is concerned with how easy it is to understand, modify and test.	Understandability, modifiability and testability
FURPS	Implicit	Adaptability, extensibility
ISO 9126	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.	Analyzability, changeability, stability, testability

We intend to distinguish software evolvability from maintainability from a collection of aspects that characterize them, such as software change stimuli that trigger the changes, type of change, impact on development process, respective focus and type of scenarios used in analysis, etc. The differences are summarized in Table 3.

Table 3 Comparisons between evolvability and maintainability

Characteristics	Evolvability	Maintainability
Software Change Stimuli	Business model, business objectives, functional and quality requirement, environment, underlying and emerging technologies, new standards, new versions of infrastructure	Defects, functional requirement, requirements from customers
Type of Change	Coarse-grained, long term, higher level, [19] radical functional or structural enhancements or adaptations	Fine-grained, short term, localized change [19]
Focus Activity	Cope with changes	Keep the system perform functions
Software Structure	Structural change	Relatively constant
Analysis Scenarios	Growth scenarios (change scenarios)	Existing use case scenarios
Development Process	May require corresponding process changes	Relatively constant
Architecture Integrity	Conformance is required	Conformance is preserved

2.3 Software Evolvability Model

Since software evolvability is a multifaceted quality attribute, we propose a software evolvability model with identification of the required sub-characteristics that a software system needs to possess in order to easily adapt to various changes during software evolution. The sub-characteristics that are identified and selected for the evolvability model are based on their importance for software developing organizations in general and their relevance for evolving software in a cost-effective way.

The process of identifying and selecting sub-characteristics is based on the earlier mentioned maintainability and evolvability analysis as well as the mentioned quality models. Evolvability-related sub-characteristics are identified and classified into six aspects. Each aspect addresses a set of quality characteristics that are covered in the well-known quality models as illustrated in Table 4. Besides, we have followed ISO 9126 standards and checked their quality attributes against our classification for completeness. Apart from the development quality attributes that are explicitly addressed in the evolvability model, the operational quality attributes, such as performance, reliability are also indirectly addressed in the sense that the improvement of these attributes are handled through e.g. analyzability and changeability. Portability and extensibility are explicit in the classification because they are essential for software evolvability. As a result, these identified sub-characteristics are relevant for evolution of software-intensive systems and cover the ranges of potential future changes that a software system may encounter during its life cycle.

Table 4 Classifications of Evolvability-Related Sub-Characteristics

Classification	Quality Characteristics in Quality Models
Analyzability	Human Engineering, Understandability
Changeability	Flexibility, Modifiability
Integrity	Reusability
Extensibility	Extensibility
Portability	Adaptability, Compatibility, Interoperability
Testability	Correctness, Efficiency

The proposed evolvability model provides a base and a catalog of check points for analyzing and evaluating software evolvability. The sub-characteristics that evolvability incorporates and their motivations are explained below.

Analyzability The capability of the software system to enable the identification of influenced parts due to change stimuli (adapted from [13]). The change stimuli include changes in business model, business objectives, functional and quality requirements, environment, underlying technologies and emerging technologies, new standards, new infrastructure, etc.

Analyzability is important since a software system must have the capability to be analyzed and explored in terms of the impact to the software by introducing a change. Many perspectives can be included in analyzability dimension, e.g. decisions on what to modify, analysis and exploration of emerging technologies from maintenance and evolution perspective, etc.

Integrity The capability of the software system to maintain architectural coherence while accommodating changes.

Integrity is a key element that may be easily ignored during software evolution. It is mostly related to understanding and coherence to the previous architectural decisions and adherence to the original architectural styles, architectural patterns or strategies. Insufficient understanding of the initial architectural constructs may have indirectly negative consequences on software structures and lead to evolvability degradation in the long run. However, taking integrity as one sub-characteristic of evolvability does not mean that the architectural constructs are not allowed to be changed. On the contrary, it helps in recognition, extraction and documentation of these architecture- related constructs as well as prevents unconscious violations against architectural principles. As a result, any necessary changes to the architecture can be conducted in a controlled way. The software architecture of an evolvable software system should allow considerable unanticipated changes in the software without compromising system integrity and invariants and can evolve in a controlled way [3].

Changeability The capability of the software system to enable a specified modification to be implemented [13].

Changeability is important since a software system must have the ease and capability to be changed without negative implications to the other parts of the software system or in a controlled way. The changeability of the software should be analyzed in correspondence to various evolution categories, e.g. new version of infrastructure or meeting business objectives. Thus, changeability is correlated to extensibility and portability in the sense that any re-factoring candidates identified in them will be eventually justified through changeability. Changeability is closely related to coupling, cohesion, modularity and software complexity in terms of software design and coding structure [14], though it is often constrained by business and economical factors.

Portability The capability of the software system to be transferred from one environment to another [13]. Portability is an example of a property that is not a sub-characteristic of maintenance but it is essential for evolvability.

Portability is one important characteristic for long term development due to the rapid technical development on hardware and software technologies. It is concerned with hardware and/or software changes, including interface and platform aspects. Therefore, it is one of the key enablers that can provide possibility to choose between different hardware and operating system vendors as well as various versions of frameworks. Portability analyses need to be made from evolution perspective, e.g. exploration of emerging technologies that may affect portability, analyzing the effect on the software architecture in terms of portability, etc.

Extensibility The capability of the software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to existing system. Extensibility is a system design principle where the implementation takes into consideration of future growth.

Extensibility is important since a software system must have the ease and capability to add on extra functionality and features, components and services to keep up with the plethora of standards, customer requirements, market requirements, etc. In order to keep its competitive edge, a software system must constantly raise the service level through supporting more functionality and providing more features [5]. This property is also characteristic for evolvability, but not for maintainability.

Testability The capability of the software system to enable modified software to be validated [13].

Testability is concerned with the verification of a software system since software modification may lead to errors and side effects, e.g. changes to one part of a system may have an unintended effect on another part of the system. Therefore, every step in the transformation and changes of software constructs need to be tested. Test cases that cover both the original and emerged changing requirements need to be identified to ensure that the system still can fulfill the original requirements and perform its intended function while meeting the new requirements.

From the list of the sub-characteristics we could assume that maintainability is a subset of evolvability, but this is only partially true. Evolvability and maintainability have different goals (changes vs. preservation as explained in Table 3) and the sub-characteristics will be evaluated in relation to these goals.

Analyzability and integrity are the center sub-characteristics and base for evolvability evaluation. The reason is that analyzability is the first core step to identify the influenced parts due to change stimuli and integrity investigation helps gain comprehensive understanding of architectural constructs related to evolvability issues of the software system, such as changeability, extensibility, portability and testability, so as to guarantee that any re-factorings made to the system will be well-planned instead of unconsciously violating existing reasonable architectural decisions.

During the software evolution process, there may be shifted focus among portability and extensibility depending on the types of emerging changes. Nevertheless, analyzability, changeability, testability and integrity are the main sub-characteristics that are required in all circumstances.

3. EVALUATING EVOLVABILITY

Software evolution and software evolvability can be examined in different phases of systems lifecycle, e.g. requirement phase, architectural phase, detailed design, and implementation and integration phases [9]. In this paper, we focus on assessing software evolvability at architectural phase. This is because software architecture is a key asset in software systems and it has tight connection to the system's quality requirements in the sense that software architectures allow or preclude nearly all of the system's quality attributes, or vice versa, the quality attributes of a software system are determined by its architecture [8].

3.1 Evaluation Method Supporting the Evolvability Model

In order to address the evolvability sub-characteristics systematically, we have extracted an approach for evolvability evaluation from an industrial case study. The application of this method and the evolvability model will be examplified in more details in a case study in section 4. The approach comprises two phases.

Phase 1: Analyze the implications of change stimuli on software architecture

This phase addresses analyzability sub-characteristics as shown in Figure 1, and includes the following two steps:

Step 1: Identify requirements on the software architecture

Step 2: Prioritize requirements on the software architecture

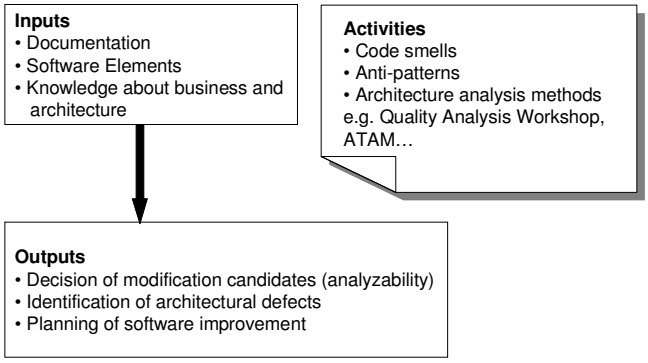

Figure 1 Software Analysis Process (Phase 1)

Phase 2: Analyze and prepare the software architecture to accommodate change stimuli and potential future changes

This phase addresses integrity, changeability, extensibility, portability and testability sub-characteristics as shown in Figure 2, and includes the following steps:

Step 3: Extract architectural constructs related to the identified issues from phase 1

Step 4: Identify re-factoring components for each identified issue

Step 5: Identify and assess potential re-factoring solutions from technical and business perspectives

Step 6: Identify and define test cases

Step 7: Present analysis results

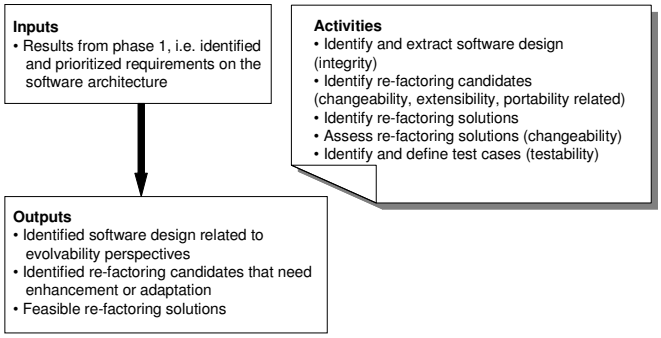

Figure 2 Software Improvement Process (Phase 2)

To summarize, the outputs of software evolvability evaluation include (i) Identified and prioritized requirements on the software architecture (ii) Established base for common understanding of these requirements from stakeholders within organizations (iii) Identified re-factoring candidates that need enhancement or adaptation (iv) Feasible re-factoring solutions.

3.2 Other Methods

There exist many architecture evaluation methods today. Some of them may be adapted to analyze software evolvability. Following is a brief description of these methods.

ATAM The Architecture Tradeoff Analysis Method (ATAM) [8] is a method for evaluating software architectures in terms of quality attribute requirements. It is used to expose the risks, non-risks, sensitivity points and trade-off points in the software architecture, therefore to achieve better architecture. It aims at different quality attributes and supports evaluation of new types of quality attributes.

SAAM The Scenario-based Architecture Analysis Method (SAAM) was originally created for evaluating modifiability of software architecture. The main outputs from a SAAM evaluation include a mapping between the architecture and the scenarios that represent possible future changes to the system, which provides indications of potential future complexity parts in the software and estimated amount of work related to the changes.

ALMA The Architecture Level Modifiability Analysis [1] is a method for analyzing modifiability based on scenarios. The outputs from an ALMA evaluation include maintenance prediction to estimate required effort for system modification to accommodate future changes, risk assessment to identify the types of changes that the system shows inability to adapt to, and software architecture comparison for optimal candidate architecture.

EBAE Empirically-Based Architecture Evaluation [16] defines a process for defining and using a number of architectural metrics to evaluate and compare different versions of architectures in terms of maintainability.

ABAS Attribute-Based Architectural Styles [4] build on architectural styles by explicitly associating with reasoning frameworks, which are based on quality attribute-specific models.

3.3 Correlations among Evaluation Methods

Among the related evaluation methods, ALMA and SAAM focus more on modifiability (changeability), EBAE on maintainability using metrics such as coupling, size and complexity, and ATAM supports multiple attributes. Since software evolvability is a multifaceted attribute, incorporating changeability among other sub-characteristics, ALMA, SAAM and EBAE will not be sufficient enough to evaluate software evolvability. Regarding ATAM, although it can support multiple quality attributes, it has one liability in dealing with future changes due to the limitation of the scenario generation process, since some evolvability scenarios may be missed which may result in wrong judgments about the current architecture [7].

The software evolvability model that we have outlined is appropriate for evolvability analysis because it pinpoints the dimensions that software architects and analysts need to consider in carrying out software evolution activities during the software evolution process. As illustrated in Figure 1, we see also the benefit of using ATAM as a basis for architecture analysis in combination with the evolvability model for evolvability evaluation.

4. CASE STUDY

The application of the proposed software evolvability model and the evaluation method was carried out on a large industrial automation system at ABB. During the long history of product development, several generations of automation controllers have been developed as well as a family of software products, ranging from programming tools to varieties of application software that

support every stage of the software system life cycle. The case study was focused on the latest generation of automation system.

4.1 Evaluated System

The software system in the automation controller today has a tremendous huge code base, consisting of several million lines of code with support for a variety of different applications and devices. All the source code is compiled into a monolithic binary software package, which has grown in size and complexity as new features and solutions are added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Besides, the software package also consists of various software applications, aiming for specific tasks that enable the automation controller to handle various applications in painting, arc welding, spot welding, gluing, machine tending or palletizing, etc.

Due to long life of products and due to continuous improvements of the products and development of new variants and new products evolvability of these systems and their components the evolvability is one of the most important properties.

4.2 Goals

The aim of the case study was to analyze software architecture of the automation system with respect to its evolvability through applying the software evolvability model. The motivations to this case study came from the emerging critical issues in terms of software evolution, which are:

- How to improve software system quality?
- How to improve the ability to enhance functionality in existing software system?
- How to build new products for dedicated market within short time?
- How to enable the ease and flexibility of distributed development of products?

Of all these questions, the root challenge is how to evaluate software evolvability and analyze whether the software system has the capability to quickly accommodate to changes. This is the necessary step towards improving software evolvability and preparing the software system for potential evolution.

4.3 Applying the Evolvability Evaluation Method

How to evolve the current monolithic automation controller software? Is it possible to evolve the controller software to meet the business objectives? We applied the software evolvability evaluation method and checked against the evolvability model to address theses issues.

Step 1: Identify requirements on the software architecture

Any change stimuli result in a collection of requirements that the software architecture needs to adapt to. The aim of this step is to extract requirements that are essential for enhancing and preparing the software architecture to cost-effectively accommodate change stimuli. Workshops and scenario-based architecture analysis methods can be used for this purpose. In our case study, several workshops were conducted for requirement identification.

The change stimuli in this case study came from the changes in business objectives, i.e. time to market, quality improvement and enabling distributed development process. The main idea to accommodate to the change stimuli was to cope with the monolithic-related issues through developing base software for domain-specific applications to build on. The base software consists of a software kernel which is the mandatory building block for all applications, as well as common extensions which are commonly used by all the applications. The base software can be packaged into software development kit, which provides necessary tools and documentation for application development. The domain-specific application parts will be separated from the base software and any application-specific extensions can be built on top of the base without the need of access to source code. This implies that the base software and domain-specific applications can be developed independently and have separate release cycles. Application developers can work more freely than before without being constrained by the release cycles of the base software. To achieve this, corresponding requirements are identified to enable the migration of monolithic architecture to modular one.

Step 2: Prioritize requirements on the software architecture

All the requirements identified from the first step need to be prioritized. In the case study, the requirements were ranked into three steps: (i) enable build of existing types of extensions, i.e. to fix all interfaces that prevent from building existing extensions after building the kernel (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects when implementing new extensions (iii) scale kernel.

Step 3: Extract architectural constructs related to the respective identified issue

In this step, we mainly focus on architectural constructs that are related to the previously identified issue. Take portability issue for example, the evaluated system is the latest generation of automation controller software, which is an evolutionary step based on earlier generations. One of the main initial design goals was to make the software portable across different target operating system (OS) platforms, as well as to run it in form of a “Virtual Controller” hosted on a general purpose computer, such as a UNIX workstation or a PC. The architecture style for the current generation automation control software is layered architecture, and within the layers object-oriented architecture. The main enabler for portability is the portability layer in the architecture. The portability layer provides interfaces for application software in the controller, including OS abstraction, POSIX file API, device driver interfaces, basic services and reusable class library. To summarize, this step is necessary to help us understand the system related to the problem issue and to discover any architectural defects around it.

Step 4: Identify re-factoring components for each identified issue

In this step, we identify the components that need re-factoring in order to fulfill the prioritized requirements. For example, in the case study, to achieve the build- and development-independency between kernel and extensions, the low-level basic services were identified as one of the re-factoring components.

Step 5: Identify and assess potential re-factoring solutions from technical and business perspectives

Technical assessment takes into consideration of change propagation and the effect of re-factoring on quality characteristics such as complexity and maintainability of the software. Business assessment estimates the cost and effort on applying re-factoring. In some cases, the solution to a certain re-

factoring component is straight forward and we know how to re-factor with local impact. Otherwise, when the implementation is uncertain and may affect several sub-systems or modules, we need to make prototype and investigate the feasibility of potential solutions as well as the estimation of implementation workload.

Step 6: Define test cases

The test cases or scenarios can be defined based on the prioritized requirements on the software architecture. Meanwhile, the software system still needs to fulfill some of the original requirements besides the new required changes. To do this, we need to identify the original test cases as well as the emerging new test cases that cover the affected component, modules or subsystems during the software evolution process. For example, in the case study, we identified test scenarios that enable separation between kernel and extension which are new test cases, and test scenarios for validating if existing domain-specific applications can still work as before without being affected after building the kernel.

Step 7: Present analysis results

The analysis results are transferred to the implementation team for further execution. In fact, the communication between analysis team and implementation team started already during the evaluation process in order to achieve mutual understanding about the re-factoring decisions.

4.4 Analysis

In this case study, we applied the evolvability model to an industrial automation controller and analyzed the software system's evolvability from a collection of dimensions. As stated in [12], software architecture that is capable of accommodating change must be specifically designed for change. Therefore, the application of the evolvability model is a necessary step in analyzing software evolvability and preparing the software system for future changes. The results of the analysis are achieved through applying the evaluation method and are presented as follows.

4.4.1 Analyzability

The knowledge of analyzability is achieved through the first two steps in the evaluation method. In this perspective, we analyze the capability of the software system to enable the identification of influenced parts due to change stimuli. The following lists the most essential activities that were required in the case study for identification of influenced parts due to change stimuli.

(1) Investigate public interfaces This improves both quality and understandability of the current system. It is error-prone to have interfaces defined as public when they should in fact be internal, e.g. application-specific software should not expose public interfaces. All public interfaces should be clearly defined and documented; including the context they can be used. In this way, there will be less and well-defined interfaces, thus to increase software quality and simplify the process of product testing.

(2) Investigate kernel and extensions This provides input to the explicit definition of the scope for kernel**,** common extensions and application-specific extensions.

(3) Investigate build dependencies The separation between kernel and extensions determines that domain-specific applications will always be built last. The build order should start from kernel, common extensions towards application extensions.

(4) Investigate impact on development process The restructuring of the automation controller software will affect the product development processes in the sense that roles, reponsibilities and working procedures, such as product interaction, verification and testing, need to respond to the change stimuli in a corresponding way.

4.4.2 Integrity

The knowledge of integrity is achieved through the third step in the evaluation method. We gained good understanding of the software architecture, although we also discovered minor violations that have taken place on the code level before the actual re-factoring work. This intensified the need of good documentation of architectural constructs and especially rationale behind each design decision.

4.4.3 Changeability

The knowledge of changeability is achieved through step 4 and 5 in the evaluation method. In this perspective, we analyze the capability of the software product to enable a specified modification to be implemented. The underlying assumptions throughout the re-factoring process in this case study were that the applied re-factoring preserves behavior and that the consistency between re-factored artifacts and other software artifacts 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.

Based on the identified re-factoring components, the respective solution and roadmap for implementation were identified and implementation workload was estimated as well. It became apparent that some modifications were easy to be implemented, while some re-factoring components may lead to considerable change propagation. It is still ongoing work to make comprehensive analysis and judgment of potential alternative solutions. Although this was the case, we found it helpful with the evaluation method to guide us through the evolvability improvement process in a structured way.

4.4.4 Extensibility

In this perspective, we analyze the capability of the software system to enhance the system with new functions and features. In the case study, it was desired that domain-specific application developers can create their own application extensions on top of the kernel software in order to respond quickly to market requirements and get rid of the tight constraints from the release cycles of the automation controller software. Therefore, the system is being prepared through executing step 4 and 5. Meanwhile, it became clear that training is necessary so that the domain-specific application developers can easily create their own applications.

4.4.5 Portability

In this perspective, we analyze the capability of the software system to be transferred in case of environment change. The portability issues in this case study include portability analysis across various target operating system platforms and portability analysis across hardware platform, thus to prepare the software system for potential environment change. It is still an on-going project around this issue, but so far, we have discovered some

aspects that need to be addressed, e.g. training for software developers in writing code that enables portability, documentation of guidelines/rules and code examples, proper use of conditional compilation in case of environment switches, etc.

4.4.6 Testability

In this perspective, we validate if the modified software system can still fulfill the original requirements as well as the new required changes. To do this, we identified emerging new test cases that cover the affected component, modules or subsystems as well as the original test cases that the software still needs to fulfill. The possibility of being able to run the program on virtual controllers simplifies a lot for testing.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a software evolvability model and an evolvability evaluation method. We contend that the evolvability of a software system can be analyzed in terms of a collection of sub-characteristics. This evolvability model is established through a systematic analysis of several existing well-known quality models and comparison analysis of distinguishable characteristics between software evolvability and maintainability. We have shown how the evolvability evaluation method and evolvability model can be applied into complex industrial context through a case study, which revealed the structured way of evaluating evolvability as well as the feasibility of using the proposed evolvability model as base and check points when evolving a software system.

Future work remains to be done to further establish the evolvability model to a hierarchical one; we need to further derive the identified sub-characteristics of evolvability to the extent when we are able to quantify them and/or make appropriate reasoning of the quality of service that a software system provides in terms of various sub-characteristic. We need to provide a catalog of guidelines and checkpoints for each sub-characteristic that can be applied in conducting evolvability analysis. We also need to analyze the correlations among the sub-characteristics with respect to constraints and trade-offs. Further we plan to establish a process framework which will enable a consistence analysis when analyzing different sub-characteristics, and when analyzing the evolvability in different phases of the product lifecycle.

6. REFERENCES

[1] Bengtsson. P. O. Architecture-Level Modifiability Analysis. Ph.D Thesis, Blekinge Institute of Technology, 2002.

[2] Bennett, K. Software Evolution: Past, Present and Future. Information and Software Technology 38 (1996) 673-680.

[3] Bennett, K. and Rajlich, V. Software Maintenance and Evolution: a Roadmap. 2000.

[4] Boehm, B. W. et al. Characteristics of Software Quality. Amsterdam, North-Holland, 1978.

[5] Bosch, J. Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach. Addison-Wesley. 2000.

[6] Brooks, F. P. No Silver Bullet. IEEE Computer, Vol. 20, No. 4, 1987.

[7] Ciraci, S. and Broek. P. Evolvability as a Quality Attribute of Software Architectures. 2003.

[8] Clements, P., Kazman, R. and Klein, M. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley. 2002.

[9] Cook, S., Ji, H. and Harrison, R. Dynamic and Static Views of Software Evolution. Proceedings IEEE International Conference on Software Maintenance ICSM, 2001.

[10] Dromey, G. Cornering the Chimera. IEEE Software (January): 33-43, 1996.

[11] Graaf, B. Maintainability through Architecture Development. EWSA, LNCS 3047, pp. 206-211, 2004.

[12] Isaac, D., McConaughy, G. The Role of Architecture and Evolutionary Development in Accommodating Change. Proc. NCOSE'94, 1994.

[13] ISO/IEC 9126-1. International Standard. Software Engineering – Product Quality – Part 1: Quality Model, 2001.

[14] ISO/IEC 9126-3. International Standard. Software Engineering – Product Quality – Part 3: Internal Metrics, 2003.

[15] Lehman, M. Laws of Software Evolution Revisited. Software Process Technology, 5th European Workshop EWSPT, 1996.

[16] McCall, J. A., Richards, P. K. and Walters, G. F. Factors in Software Quality. National Technical Information Service, 1977.

[17] Ortega, M, et al. Construction of a Systemic Quality Model for Evaluating a Software Product. Software Quality Journel, v11, n3, p219-42, Sept 2003.

[18] Pfleeger, S. L. The Nature of System Change. IEEE Software, 1998.

[19] Weiderman. N. H. et al. Approaches to Legacy Systems Evolution. Technical Report CMU/SEI-97-TR-014, 1997.

[20] Yang, H. and Ward, M. Successful Evolution of Software Systems. Artech House Publishers, London, 2003