# Using Software Evolvability Model for Evolvability Analysis

Hongyu Pei Breivold
*ABB Corporate Research*
*Industrial Software Systems*
*SE-721 78 Västerås, Sweden*
*Hongyu.pei-breivold@se.abb.com*

Ivica Crnkovic
*Mälardalen University*
*School of Innovation, Design and Engineering*
*SE-721 23 Västerås, Sweden*
*Ivica.crnkovic@mdh.se*

## Abstract

*Software evolution is characterized by inevitable changes of software and increasing software complexities, which in turn may lead to huge costs unless rigorously taking into account change accommodations. This is in particular true for long-lived systems in which changes go beyond maintainability. For such systems, there is a need to address evolvability explicitly in the requirements and early design phases and maintain it during the entire lifecycle. Nevertheless, there is a lack of a model that can be used for analyzing, evaluating and comparing software systems in terms of evolvability. In this paper, we describe the initial establishment of an evolvability model as a framework for analysis of software evolvability. We motivate and exemplify the model through an industrial case study of a software-intensive automation system.*

## 1. Introduction

Software maintenance and evolution are characterised by their huge cost and cumbersome implementation [1, 2]. 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, software evolvability must be considered. However, 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. Besides, the lack of evolvability model hinders us from analyzing and evaluating software systems in terms of evolvability.

In this paper, we intend to (i) outline the differences between software maintainability and evolvability, (ii) define a software evolvability model and (iii) identify subcharacteristics of software evolvability based on literature surveys, analyses of several well-known quality models and an industrial case study. 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 through an industrial case study, in which evolvability was analyzed.

The outline of the paper is as follows: Section 2 describes the related work. In section 3, we analyze the differences between evolvability and maintainability. Section 4 presents the motivations for evolvability analysis from the case perspective. We outline a software evolvability model in section 5, where necessary subcharacteristics of software evolvability and corresponding measuring attributes are identified. Further in section 6, we present the structured way of evolvability evaluation that we used in the case study, and demonstrate the evolvability model with following analysis. Section 7 concludes the paper and outlines the future work.

## 2. Related Work

Several metrics have been proposed for evaluating evolvability. Ramil and Lehman proposed metrics

based on implementation change logs [26] and computation of metrics using the number of modules in a software system [19]. Another set of metrics is based on software life span and software size [30]. In [29], a framework of process-oriented metrics for software evolvability was proposed to intuitively develop architectural evolvability metrics and to trace the metrics back to the evolvability requirements based on the NFR framework [6].

An approach was described in [20] to measure software architecture's quality characteristics through identified key use cases, based on the customization of the ISO 9126 standard. An ontological basis which allows for the formal definition of a system and its change at the architectural level is presented in [27] and applied to the domain of computer-based systems engineering. Taxonomy was proposed in [28] to address change as factors and classify evolvability into several aspects, e.g. generality, adaptability, scalability and extensibility. However, it does not cover all the types of software evolution, e.g. concerns of product line development.

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 [22], Boehm [3], FURPS [13], ISO 9126 [16] and Dromey [10]. Although several quality attributes are correlated to software evolvability, e.g. extensibility and maintainability, the term evolvability is not explicitly addressed in either of the quality models.

## 3. Evolvability vs. Maintainability

In this paper, we use evolution to refer to the particular evolution stage as described in the staged model by Bennett and Rajlich [2]. Among the various definitions of evolvability [7, 14, 24, 28], we refer to the definition in [28], since it expresses the dynamic behaviour during a software system's lifecycle and supports the staged model:

*"Evolvability: An attribute that bears on the ability of a system to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity."*

Evolvability is, similar to maintainability, a system characteristic that depends on many subcharacteristics. The problem of evolvability is more difficult than maintainability; in evolvability we should expect the unexpected changes. However, many people use software evolvability as synonymous to software maintainability. Therefore, we give a summary of the definitions of maintainability in various quality models in Table 1, and summarize the differences between evolvability and maintainability in Table 2. We intend to distinguish software evolvability from maintainability from a collection of aspects, 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.

**Table 1 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 |

**Table 2 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 | Bugs, functional requirement, requirements from customers |
| **Type of Change** | Coarse-grained, long term, higher level, [31] radical functional or structural enhancements or adaptations | Fine-grained, short term, localized change [31] |
| **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 |

## 4. Motivations for Evolvability Analysis from a Real Case

The need to explicitly address software evolvability is becoming recognized [7]. This is in particular true for long-lived systems in which changes go beyond maintainability. We have seen at ABB examples of different industrial systems that often have a lifetime of 20-30 years. These systems are subject to and may undergo a substantial amount of evolutionary changes, e.g. software technology changes, software systems merge due to organizational changes, demands for distributed development, system migration to product line architecture, etc.

The evolution problems we have observed came from different cases. In this paper, we exemplify and analyze in particular one industrial case study that was carried out on a large automation control 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. The case study focused on the latest generation of the software controller.

The controller software consists of more than three million lines of code written in C/C++ and a complex threading model, with support for a variety of different applications and devices. It has grown in size and complexity, as new features and solutions have been added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Such a complex system is difficult to maintain. It is also important and considerably more difficult to evolve. Due to different measures such as organizational and lifecycle process improvements, the system keeps the maintainability, but the evolvability becomes more difficult since the increased complexity in turn leads to decreased flexibility, resulting in problems to add new features. Consequently, it would become costly to adapt to new market demands and penetrate new markets.

Our particular system is delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications aim for specific tasks in painting, arc welding, spot welding, gluing, machine tending and palletizing, etc. In order to keep the integration and delivery process efficient, the initial architectural decision was to keep the deployment artifact monolithic; The complete set of functionality and services is present in every product even though not everything is required in the specific product. As the system grew, it became more difficult to ensure that the modifications of specific application software do not affect the quality of other parts of the software system. As a result, it becomes difficult and time-consuming to modify software artifacts, integrate and test products. To continue exploiting the substantial software investment made and to continuously improve the system for longer productive lifetime, it has become essential to explicitly address evolvability, since the inability to effectively and reliably evolve software systems means loss of business opportunities [2]. We want to emphasize here that the problem raised is not a problem of maintainability. The major problems arise when broad new (very different) features or different development paradigms, shifting business and organizational goals are introduced, so the problems related to the software evolvability – a fundamental element for increasing strategic and economic value of the software [31].

To solve the problems presented above, we need to handle several research issues: (i) which characteristics are necessary for a software system to be evolvable; (ii) how to assess evolvability in a systematic manner; (iii) how to achieve evolvability; and (iv) how to measure evolvability. We will address these issues in section 5 and 6.

## 5. Software Evolvability Model

Software evolvability concerns both business and technical issues [32], since the stimuli of changes come from both perspectives, e.g. environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software structures and/or functionality.

Software evolution and software evolvability can be examined at different levels of software systems, e.g. requirement level, architectural level, detailed design, and source code level [9]. In this paper, we focus on assessing software evolvability at architectural level. This is because software architecture is a key asset in a software system, which allows or precludes nearly all of the system's quality attributes [8].

### 5.1. Evolvability Model

Software evolvability is a multifaceted quality attribute [28]. Based on the definition of evolvability in [28], the software quality challenges and assessment [12, 14], the types of change stimuli and evolution [5], and experiences we gained through industrial case studies, we have discovered that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [16] is not sufficient for a software system to be evolvable. Therefore, we have (i) complimented and identified

subcharacteristics that are of primary importance for an evolvable software system, and (ii) outlined a software evolvability model that provides a basis for analyzing and evaluating software evolvability.

The idea with the evolvability model is to further derive the identified subcharacteristics to the extent when we are able to quantify them and/or make appropriate reasoning about the quality of service, as in Figure 1.

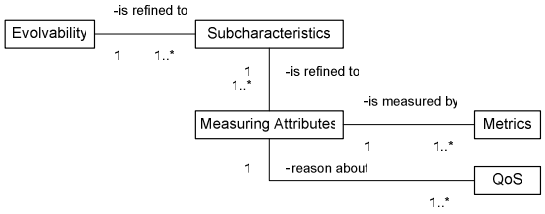


**Figure 1 Concept of the evolvability model**

The identified subcharacteristics are summarized in Table 1. They are a union of quality characteristics having to do with changes, and are relevant for characterization of evolution of software-intensive systems during their life cycle. With these subcharacteristics in mind, we have a basis on which different systems can be examined and compared in terms of evolvability. Any system that does not explicitly address one or more of these subcharacteristics is missing an element that probably will undermine the system's ability to be evolved.

**Table 3 Subcharacteristics of evolvability**

| Sub-characteristics | Description |
|---|---|
| Analyzability | The capability of the software system to enable the identification of influenced parts due to change stimuli (based on [16]). |
| Integrity | The non-occurrence of improper alteration of architectural information (based on [18]). |
| Changeability | The capability of the software system to enable a specified modification to be implemented and avoid unexpected effects (based on [16]). |
| 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 the existing system (based on [16]). |
| Portability | The capability of the software system to be transferred from one environment to another [16]. |
| Testability | The capability of the software system to enable modified software to be validated [16]. |
| Domain-specific attributes | The additional quality subcharacteristics that are required by specific domains [12]. |

These subcharacteristics serve as a catalog of check points for evaluation, and each subcharacteristic is motivated and explained below in conjunction with the case study. Examples of measuring attributes for each subcharacteristic are given. However, the description of how to apply the measuring attributes through metrics is subject for further research.

**Analyzability**

*Case Motivation*: The release frequency of the controller software is twice a year, with around 40 various new requirements that need to be implemented in each release. These requirements may have impact on different attributes of the system, and the possible impact must be analyzed before the implementation of the requirements. This requires that the software system must have the capability to be analyzed and explored in terms of the impact to the software by introducing a change.

*Description*: Many perspectives are included in this dimension, e.g. identification and decisions on what to modify, analysis and exploration of emerging technologies from maintenance and evolution perspectives.

*Measuring attributes*: modularity, complexity, documentation.

**Integrity**

*Case Motivation*: A strategy for communicating architectural principles that we found out from various case studies was to appoint members of the core architecture team as technical leaders in the development projects. However, this strategy although helpful to certain extent, did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions, resulting in unconscious violation of architectural conformance. This may lead to evolvability degradation in the long run.

*Description*: Architectural integrity is related to understanding and coherence to the architectural decisions and adherence to the original architectural styles, patterns or strategies. However, taking integrity as one subcharacteristic of evolvability does not mean that the architectural approaches are not allowed to be changed. Proper architectural integrity management is essential for the architecture to allow unanticipated changes in the software without compromising software integrity and to evolve in a controlled way [2].

*Measuring attributes*: architectural documentation.

**Changeability**

*Case Motivation*: Due to the monolithic characteristic of the controller software, modifications in certain parts of the software package may lead to ripple effects, and requires recompiling, reintegrating and retesting of the whole system. This results in inflexibility of patching and customers have to wait for a new release even in case of corrective maintenance and configuration changes. Therefore, it is strongly required that the software system must have the ease and capability to be changed without negative implications or with controlled implications to the other parts of the software system.

*Description*: Software architecture that is capable of accommodating change must be specifically designed

for change [15]. Changeability is closely related to coupling, cohesion, modularity and software complexity in terms of software design and coding structure [17, 23].
*Measuring attributes*: complexity, coupling, change impact, encapsulation, reuse, modularity.

**Portability**
*Case Motivation*: The current controller software supports VxWorks and Microsoft Windows NT. There is a need of openness for choosing among different operating system vendors, e.g. Linux and Windows CE.
*Description*: Due to the rapid technical development on hardware and software technologies, portability 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.
*Measuring attributes*: mechanisms facilitating adaptation to different environments.

**Extensibility**
*Case Motivation*: The current controller software supports around 20 different applications that are developed by several distributed development centers around the world. To adapt to the increased customer focus on specific applications and to enable establishment of new market segments, the controller, like any other software systems, must constantly raise the service level through supporting more functionality and providing more features [4].
*Description*: One might argue that extensibility is a subset of changeability. Due to the fact that about 55% of all change requests are new or changed requirements [25], we define extensibility explicitly as one subcharacteristic of evolvability. It is a system design principle where the implementation takes future growth into consideration.
*Measuring attributes*: modularity, coupling, encapsulation, change impact.

**Testability**
*Case Motivation*: The controller software exposed huge number of public interfaces which resulted in tremendous time merely on interface tests. One task was therefore to reduce the public interfaces to around 10%. Besides, due to the monolithic characteristic, error corrections in one part of the software requires retesting of the whole system. One issue was therefore to investigate the feasibility of testing only modified parts.
*Description*: According to statistics [11], software testing spends as much as 50% of development costs and comprises up to 50% of development time. Hence, testability is a key feature permitting high quality to be combined with reduced time-to-market.
*Measuring attributes*: complexity, modularity.

**Domain- specific attributes**
*Case Motivation*: The controller software has critical real-time calculation demands and is expected to reduce base software code size and runtime footprint. Besides, the devices that the controller software supports are required to have a MTBF (Mean Time Between Failures) with up to 50000 hours.
*Description*: Different domains may require additional quality characteristics that are specific for a software system to be evolvable. For example, the World Wide Web domain requires additional quality characteristics such as visibility, intelligibility, credibility, engagibility and differentiation [12]. Component exchangeability in the context of service reuse [21] is another example within the distributed domain, e.g. wireless computing, component-based and service-oriented applications.
*Measuring attributes* depend on the specific domains.

# 6. Case study

The change stimuli to the controller software in the case study came from the emerging critical issues in terms of software evolution, which are: (i) time-to-market requirements, such as building new products for dedicated market within short time; (ii) improvement of software system quality; (iii) increased ease and flexibility of distributed development of products in combination with the diversity of application variants.

## 6.1. Evaluating Evolvability

In order to address evolvability, we conducted the following structured evaluation steps as shown in Figure 2.

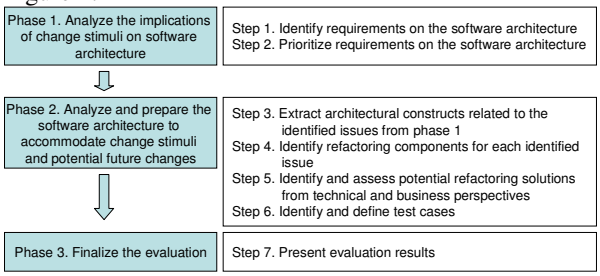


**Figure 2 Evaluation steps**

***Phase 1****: Analyze the implications of change stimuli on software architecture.*
This phase addresses analyzability.

***Step 1:*** *Identify requirements on the software architecture.*

Any change stimulus results 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 software architecture enhancement so as to cost-effectively accommodate to change stimuli. Several architecture workshops were conducted, where the architecture core team members and key stakeholders met and identified the following requirements on the software architecture.

**R1.** The software architecture needs to be migrated from monolithic to modular one.

**R2.** The complexity of the architecture structures needs to be reduced.

**R3.** The architecture needs to enable distributed development of extensions with minimum dependency.

**R4.** The portability needs to be investigated.

**R5.** The impact on product development process needs to be investigated.

**R6.** The base software code size and runtime footprint need to be reduced.

***Step 2:*** *Prioritize requirements on the software architecture.*

In order to establish a basis for common understanding of architecture requirements among the stakeholders within the organization, all the requirements identified from the first step were prioritized. Since the main idea was to apply product line approach and separate application-specific extensions from base software, the criteria for requirement prioritization were: (i) enable building of existing types of extensions after refactoring and architecture restructuring (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects on implementing new extensions.

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

This phase addresses integrity, changeability, extensibility, portability, testability and domain-specific attributes. Mappings must be made between each identified requirement and the perceived evolvability subcharacteristics. This is used to check the model and the requirements for consistency and completeness.

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

We mainly focused on architectural constructs that are related to each identified requirement from integrity perspective. Integrity management is illustrated in Figure 3. An architectural approach comprises of its intent, applicability, affected quality attributes and supported generic or specific scenarios. During software maintenance and evolution, an emerging architectural approach introduces certain consequences on the quality and behavior of the software system. Furthermore, it might be in conflict with existing architectural approaches. In this case, an evaluation is required to verify the appropriateness of existing and emerging architectural approaches, and identify design tradeoffs if any. In this sense, documentation and evaluation of architectural approaches play a key role in integrity management.

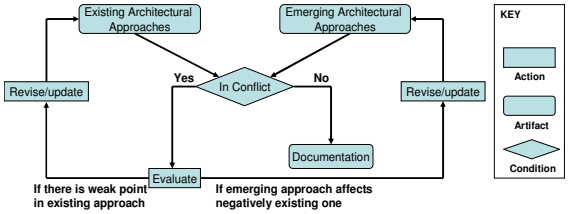


**Figure 3 Integrity management**

***Step 4:*** *Identify refactoring components for each identified issue.*

In this step, we identified the components that need refactoring in order to fulfill the prioritized requirements. For example, the basic idea of architecture restructuring in the case study was to divide the architecture into three parts: a kernel, common extensions and application-specific extensions. To achieve the build- and development-independency between the kernel and extensions, functionality needed to be separated from resource management. Accordingly, the low-level basic services in resource allocations were identified as some of the components that need refactoring, e.g. semaphore ID management component, memory allocation management component.

***Step 5:*** *Identify and assess potential refactoring solutions from technical and business perspectives.*

The change propagation of the effect of refactoring was considered and was provided as an input to the business assessment, estimating the cost and effort on applying refactorings. In some cases, the refactoring of a certain component was straight forward and we knew how to refactor with only local impact. When the implementation was uncertain and might affect several subsystems or modules, prototypes were made to investigate the feasibility of potential solutions as well as the estimation of implementation workload.

***Step 6:*** *Identify and define test cases.*

The emerging new test cases that cover the affected component, modules or subsystems were identified. We identified regression test scenarios that were used

to test the separation between kernel and extension, and test scenarios for validating if existing domain-specific applications can still work as intended without being affected after building the kernel. A test scenario example is to implement an additional option that contains a task utilizing the basic services.

***Phase 3**: Finalize the evaluation.*
***Step 7**: Present evaluation results.*

The evaluation results included (i) the identified and prioritized requirements on the software architecture; (ii) identified components/modules that need to be refactored for enhancement or adaptation; (iii) refactoring investigation documentation which describes the current situation and solutions to each identified candidate that need to be refactored, including estimated workload; and (iv) test scenarios.

## 6.2. Analysis of Software Evolvability Subcharacteristics

The requirements that needed to be addressed in the case study were captured and mapped towards the evolvability subcharacteristics as shown in Table 4.

**Table 4 Subcharacteristics and requirements mapping**

| Subcharacteristics | Requirements |
|---|---|
| Analyzability | R1 – R6 |
| Integrity | R1 – R6 |
| Changeability | R1. The software architecture needs to be migrated from monolithic to modular one |
| Extensibility | R3. The architecture needs to enable distributed development of extensions with minimum dependency |
| Portability | R4. Portability needs to be investigated |
| Testability | R5. The impact on product development process needs to be investigated |
| Domain-specific attributes | R6. The base software code size and runtime footprint need to be reduced |

The results of the evolvability analysis were achieved through applying the aforementioned evaluation steps and are presented below.

### 6.2.1. Analyzability.

The knowledge of analyzability is achieved through the first two steps (Figure 2). The activities for each identified requirement were refined:

**R1.** The software architecture needs to be migrated from monolithic to modular one:

a) Enable the separation of layers within the controller software: (i) a kernel which consists of components that must be included by all application variants; (ii) common extensions which are available to and can be selected by all application variants; and (iii) application extensions which are only available to specific application variants.
b) Investigate dependencies between the existing extensions.

**R2.** The architecture complexity needs to be reduced:

a) Define interfaces and reduce public interface calls.
b) Support task isolation and task management.
c) Support choosing a new scripting language, since modern scripting languages are flexible, productive and reduce the need to recompile.

**R3.** The architecture needs to enable distributed development of extensions with minimum dependency:

a) Build the application-specific extensions on top of the base software (kernel and common extensions) without the need of access to the internal base source code.
b) Investigate existing dependencies between base software and application extensions.
c) Package the base software into software development kit, which provides necessary interfaces, tools and documentation to support distributed application development.
d) Separate release cycles of the base software and application-specific extensions.

**R4.** The portability needs to be investigated:

a) Investigate portability across various target operating system platforms.
b) Investigate portability across hardware platforms.

**R5.** The impact on product development process needs to be investigated:

a) Investigate the implications of restructuring the automation controller software, with respect to product integration, verification and testing.

**R6.** The base software code size and runtime footprint need to be reduced:

a) Investigate enabling mechanisms, e.g. properly partitioning functionality.

### 6.2.2. Integrity.

The knowledge of integrity was achieved through the third step (Figure 2). Over years of development, a lot of functionality has been added to the system to support new requirements. It becomes easy to unconsciously violate the original good design decisions, especially when there is a lack of proper tool support to monitor the violations, e.g. improper use of conditional compilation in case of environment changes. To prevent any implementation violations, two aspects were considered: (i) Extract design decisions through documentation of architectural constructs, with especially rationale specified for each design decision. An example of documenting rationale for a portability-related architectural design decision is illustrated in Table 3. (ii) Provide training, guidelines/rules and code examples for software developers in writing code and using tactics that

enable the achievement of a certain quality characteristic.

**Table 5 Documentation of rationale for a design decision**

| | |
|---|---|
| Rationale ID | Portability R-id1 |
| Architectural design decision | Encapsulate COTS infrastructure technology choices, including operating systems and communication services |
| Candidate alternatives | Introduce portability layer |
| Consequences | Easy to move to a different operating system and hardware platform (+)<br>Encapsulation of technology choices (+)<br>Tradeoff against performance (-) |

In order to maintain and evolve the architecture for future development, tool support such as Lattix LDM (www.lattix.com) was one alternative. It can be used for defining design rules, e.g. rules for against direct access to OS native APIs, rules to indicate software exposing hardware-content dependencies, rules to reflect layering architecture, etc. These rules allow for the periodic automatic checking of the code base for design violations.

### 6.2.3. Changeability.

The knowledge of changeability was achieved through step 4 and 5 (Figure 2). To cope with R1, consistent changes needed to be carried out to restructure the original function-oriented architecture to product-line architecture as illustrated in Figure 4.

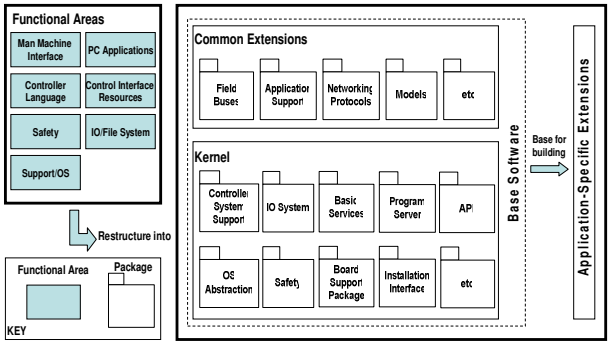


**Figure 4 Architecture restructuring**

The product line approach was adopted to achieve clear separation of concerns and interface definition from two perspectives: (i) Identify commonalities e.g. modules, components and services that are essential for all applications, and exclude those that are bound to a specific application. Figure 5 illustrates the dependency analysis between specific applications/common extensions and base services, where x represents the expected presence of a dependency and nothing for its absence. It is not a complete list due to company confidentiality. (ii) Identify dependencies between existing applications and plan for future potential dependencies. Some applications have dependencies because they need to be run on the same controller. Therefore, sufficient control of product features is required.

| | Services | | | | | |
|---|---|---|---|---|---|---|
| **Application Extensions** | alarm | error log | ipc | event | device | etc |
| Arc welding | x | x | x | | | |
| Painting | x | x | x | | | |
| Picking, Packing | x | x | x | | | |
| etc | | | | | | |
| **Common Extensions** | | | | | | |
| Field buses | | x | | | x | |
| Application support | x | x | | | | |
| Models | x | x | x | x | | |
| etc | | | | | | |

**Figure 5 Dependency analysis extracting kernel/extension**

To cope with the above two perspectives, corresponding refactoring work needed to be done. All the ripple effects must be investigated. In this sense, tool support was necessary for building dependency structure matrix and creating what-if and should-be architectures.

### 6.2.4. Extensibility.

The knowledge of extensibility was achieved through step 4 and 5 (Figure 2). To cope with R3, a Base Software SDK (Software Development Kit), consisting of the kernel and common extensions, should be developed, thus to enable distributed development of extensions. The SDK includes well-documented API (Application Programming Interface), wizards and tools for developing application-specific extensions. Accordingly, separate release cycles for base software and applications become possible due to the clear separation of concerns after the architecture restructuring. To minimize negative side-effect of extensions on the behavior and quality of the final system, the fault-tolerant mechanisms for extensions need to be considered, e.g. the possibility of dividing software into multiple containment zones, resource and functionality isolation, thread management, etc.

### 6.2.5. Portability.

The knowledge of portability was achieved through step 4 and 5 (Figure 2). One of the main 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 as shown in Figure 6.

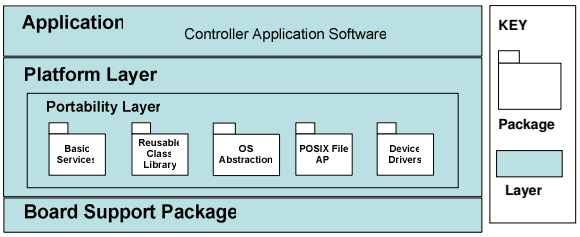


**Figure 6 Layered architecture in portability perspective**

The portability layer encapsulates many infrastructure technology choices and provides interfaces for application software in the controller. Within the platform layer, there are also other layers, which are not shown in the figure.

#### 6.2.6. Testability.

The knowledge of testability was achieved through step 6 (Figure 2). Since the kernel merely contains a set of loosely coupled components but no complete applications, a test application having sufficient functionality is needed to ensure that the kernel covers sufficient functionality for applications to build on. Adopting the product line approach, the amount of variability has to be limited to avoid system complexity and decreased testability. Tool support was needed, e.g. appropriate regression testing frameworks and AQTime (www.automatedqa.com) for code coverage. One technical challenge is to investigate model-based verification and enable testing of certain quality properties of extensions.

#### 6.2.7. Domain-specific attributes

The knowledge of domain-specific attributes is achieved through step 4 and 5 (Figure 2). To cope with R6, we need to partition the functionality of the controller software and create a Base Software SDK that contains separate link modules to build products. A constraint with respect to reducing the runtime footprint is the size of memory footprint that each session consumes, e.g. memory used per deployed component and per connection. This is a work-in-progress.

## 7. Conclusions and Future Work

Based on literature and industrial case studies, we identify subcharacteristics that are of primary importance for evolvable software systems and outline a software evolvability model. We exemplify with a case study on how the model can be applied into complex industrial context to assist software evolvability analysis, with the aid of the structured evaluation steps. All involved stakeholders expressed that they were pleased with this systematic approach, as it made architecture requirements and corresponding design decisions more explicit, better founded and documented. By establishing the evolvability model, we hope to have improved the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus. This, in turn, helps us to prolong the evolution stage.

We intend to continue working on the evolvability model by conducting more case studies to confirm and refine the model. We need to further explore the application of the evaluation steps, and generalize towards an evolvability evaluation method. Meanwhile, we need to provide a catalog of guidelines for each subcharacteristic that can be applied in conducting evolvability analysis. Further we plan to analyze the correlations among the subcharacteristics with respect to constraints and tradeoffs.

## 8. References

[1] K. Bennett. Software Evolution: Past, Present and Future. Information and Software Technology 38 (1996) 673-680.
[2] K. Bennett and V. Rajlich. Software Maintenance and Evolution: a Roadmap. The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press 2000.
[3] B.W. Boehm et al. Characteristics of Software Quality. Amsterdam, North-Holland, 1978.
[4] J. Bosch. Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach. Addison-Wesley. 2000.
[5] N. Chapin et al. Types of Software Evolution and Software Maintenance, Journal of Software Maintenance and Evolution: Research and Practice, 2001.
[6] L. Chung et al. Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Boston, 2000.
[7] S. Ciraci and P. Broek. Evolvability as a Quality Attribute of Software Architectures. The International ERCIM Workshop on Software Evolution 2006.
[8] P. Clements, R. Kazman and M. Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley. 2002.
[9] S. Cook, H. Ji and R. Harrison. Dynamic and Static Views of Software Evolution. Proceedings IEEE International Conference on Software Maintenance ICSM, 2001.
[10] G. Dromey. Cornering the Chimera. IEEE Software (January): 33-43, 1996.
[11] N.S. Eickelmann and D.J. Richardson. What Makes One Software Architecture More Testable Than Another? SIGSOFT Workshop, 1996.
[12] R. Fitzpatrick et al. Software Quality Challenges. 26th International Conference on Software Engineering, 2004.
[13] R. Grady and D. Caswell. Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs, NJ, PrenticeHall. 1987.
[14] R.F. Hilliard et al. MITRE's Architecture Quality Assessment, Software Engineering and Economics Conference, 1997.

[15] D. Isaac and G. McConaughy. The Role of Architecture and Evolutionary Development in Accommodating Change. Proc. NCOSE'94, 1994.
[16] ISO/IEC 9126-1. International Standard. Software Engineering – Product Quality – Part 1: Quality Model, 2001.
[17] ISO/IEC 9126-3. International Standard. Software Engineering – Product Quality – Part 3: Internal Metrics, 2003.
[18] Laprie, Dependable Computing and Fault-Tolerant Systems. Vol. 5, Dependability: Basic Concepts and Terminology in English, French, German, Italian, and Japanese. Laprie, J.C. (ed.). New York: Springer-Verlag, 1992
[19] M.M. Lehman and J.F. Ramil et al. Metrics and Laws of Software Evolution – The Nineties View. IEEE Computer Press, pp 20-32, 1997.
[20] F. Losavio et al. ISO Quality Standards for Measuring Architectures. The Journal of Systems and Software, 2004.
[21] C. Lüer et al. The Evolution of Software Evolvability. IWPSE 2001.
[22] J.A. McCall, P.K. Richards and G.F. Walters. Factors in Software Quality. National Technical Information Service, 1977.
[23] M. Ortega et al. Construction of a Systemic Quality Model for Evaluating a Software Product. Software Quality Journel, v11, n3, p219-42, Sept 2003.
[24] G.S. Percivall. System Architecture for Evolutionary System Development, Proc. NCOSE'94.
[25] T.M. Pigoski. Practical Software Maintenance. Wiley Computer Publishing, 1996.
[26] J.F. Ramil and M.M. Lehman. Metrics of Software Evolution as Effort Predictors – A Case Study. Proceedings of the International Conference on Software Maintenance, 2000.
[27] D. Rowe and J. Leaney. Evaluating Evolvability of Computer Based Systems Architectures – an Ontological Approach. Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems, 1997.
[28] D. Rowe, J. Leaney and D. Lowe. Defining Systems Evolvability – a Taxonomy of Change. Proceedings of the IEEE Conference on Computer Based Systems, 1998.
[29] N. Subramanian and L. Chung. Process-Oriented Metrics for Software Architecture Evolvability. Proceedings of the 6th International Workshop on Principles of Software Evolution, 2002.
[30] T. Tamai and Y. Torimitsu. Software Lifetime and its Evolution Process over Generations. Proceedings of the International Conference on Software Maintenance, 1992.
[31] N.H. Weiderman et al. Approaches to Legacy Systems Evolution. Technical Report CMU/SEI-97-TR-014, 1997.
[32] H. Yang and M. Ward. Successful Evolution of Software Systems. Artech House Publishers, London, 2003