Using Dependency Model to Support Software Architecture Evolution

Hongyu Pei Breivold[1], Ivica Crnkovic[2], Rikard Land[2], Stig Larsson[3]
[1]*ABB Corporate Research, Industrial Software Systems, 721 78 Västerås, Sweden*
hongyu.pei-breivold@se.abb.com

[2]*Mälardalen University, 721 23 Västerås, Sweden*
{ivica.crnkovic, rikard.land}@mdh.se

[3]*ABB AB, 721 78 Västerås, Sweden*
stig.bm.larsson@se.abb.com

Abstract

Evolution of software systems is characterized by inevitable changes of software and increasing software complexity, which in turn may lead to huge maintenance and development costs. For long-lived systems, there is a need to address and maintain evolvability (i.e. a system's ability to easily accommodate changes) during the entire lifecycle. As designing software for ease of extension and contraction depends on how well the software structure is organized, this paper explores the relationships between evolvability, modularity and inter-module dependency. Through a case study of an industrial power control and protection system, we describe our work in managing its software architecture evolution, guided by the dependency analysis at the architectural level. The paper includes also the main analysis results, our experiences and reflections during the dependency analysis process in the case study.

1. Introduction

The role of software architecture in the evolution of software-intensive systems is being recognized and becoming increasingly important, as software architecture allows or precludes nearly all of the system's quality attributes [2, 11]. The evolution of software architecture implies integrating changing requirements and coping with stakeholders' concerns with respect to business, technology, process and organizational perspectives, which in turn may result in increased complexity. These phenomena of continuous change and increasing complexity in software systems were recognized by Lehman and expressed in his laws of software evolution [23]. In addition, one property of software systems noted by Brooks [5] is invisibility of software structure representation, which further negatively affects the software architecture evolution. Therefore, a lot of research has been done in exploring the relationship between the design of a complex system and the manner in which this system evolves over time [27]. We describe in our earlier work [34] an evolvability model which refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes. This paper is a continuation of our earlier work [34] and further explores one particular measuring attribute, i.e. modularity, which affects the behavior of a design with respect to most of the evolvability subcharacteristics, as designing software for ease of extension and contraction depends on how well the software structure is organized and modular designs are argued to be more evolvable [27, 33], i.e. these designs facilitate making future adaptations. Although the value of modularity has been long recognized [41], not much data has been published with respect to large scale industrial software systems [22]. To enrich the knowledge in this direction, we describe our experiences through an industrial case study, with respect to (i) exploring the relationship between software evolvability, modularity and inter-module dependencies; (ii) using dependency model to support software architecture evolution; and (iii) to share industrial software evolution experiences with respect to reflections from the dependency analysis process.

The remainder of this paper is structured as follows. Section 2 summarizes our evolvability model and in particular explores the relationship between software evolvability, modularity and inter-module dependencies. Section 3 presents the methodology that we used in the case study. Section 4 presents the case study of an industrial control and protection software system and describes our work in managing the software architecture evolution through dependency analysis. Section 5 discusses the experiences we gained through the case study. Section 6 reviews related work and finally section 7 concludes the paper.

2. Evolvability, Modularity and Inter-Module Dependencies

This section summarizes first the evolvability model from our earlier work [34] and secondly, explores further the relationships between modularity, evolvability subcharacteristics and inter-module dependencies.

2.1 Evolvability Model

Software evolvability is a multifaceted quality attribute [35]. Based on the definition of evolvability in [35], analysis of various quality models [4, 13, 16, 21, 29], the software quality challenges and assessment [15], the types of change stimuli and evolution [9], and experiences we gained through industrial case studies, we have identified subcharacteristics that are of primary importance for an evolvable software system, and 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.

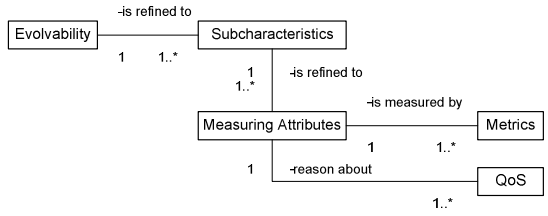

Figure 1 Elements of the evolvability model

The subcharacteristics and examples of their measuring attributes described in [34] are summarized in Table 1. Definitions of these subcharacteristics are provided in section 2.2. Failing in achieving any of these subcharacteristics probably will undermine the system's ability to be evolved.

Table 1 Subcharacteristics of evolvability and measuring attributes

Subcharacteristics	Measuring Attribute
Analyzability	modularity, complexity, documentation
Architectural Integrity	architectural documentation
Changeability	modularity, complexity, coupling, change impact, encapsulation, reuse
Extensibility	modularity, coupling, encapsulation, change impact
Portability	mechanisms facilitating adaptation to different environments
Testability	modularity, complexity
Domain-specific attributes	depend on the specific domains

2.2 Modularity and Subcharacteristics of Evolvability

This section explains the relationship between modularity and evolvability subcharacteristics. Modularity is a concept by which a piece of software is grouped into a number of distinct and logically cohesive subunits, presenting services to the outside world through a well-defined interface [12]. Modularization is a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time [32].

Modularity and analyzability Analyzability is the capability of the software system to enable the identification of influenced parts due to change stimuli, such as changes in environment, organization, process, technology and stakeholders' needs. Modularity plays an important role because an analysis of independent modules in isolation is easier to perform than in an analysis where a module is heavily dependent on other modules. Components that have excessive and unexpected dependencies are hard to work with because they cannot be understood easily in isolation. Statistics show that between 50% and 90% of software maintenance involves the understanding of the software being maintained [40], which implies the essence of modularity to achieve software analyzability.

Modularity and architectural integrity Architectural integrity is the non-occurrence of improper alteration of architectural information. A direct connection between modularity and architectural integrity does not exist. However, the modularization mechanisms and techniques, tactics and rationale for each design choice need to be documented to ensure architectural integrity. This documentation process is essential for the architecture to allow unanticipated changes in the software without compromising software integrity and to evolve in a controlled way [3].

Modularity and changeability Changeability is the capability of the software system to enable a specified modification to be implemented and avoid unexpected effects. Modularity plays an important role in software changeability because it reduces the probability that a change to one module propagates to other modules, and vice versa, to keep outside modifications from propagating into the module. According to [2], modularity increases the range of manageable complexity and accommodates uncertainty. Components that have excessive and unexpected dependencies are hard to work with because changes to functionality cannot be easily localized. Modularity determines software quality in terms of changeability [18]. Complex relationships between components

make it difficult to anticipate and identify the ripple effects of changes [14].

Modularity and extensibility Extensibility is 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. Modularity plays an important role in extensibility because it supports separating concerns and enables definition of extension points [10] based on such considerations as coupling, cohesion. Components that have excessive and unexpected dependencies are hard to work with because the impact of extensions to functionality cannot be easily localized, and may adversely impact the capability of the software system to handle future additions without the need to rewrite existing functionality.

Modularity and portability Portability is the capability of the software system to be transferred from one environment to another. Modularity plays an important role in portability because it enforces information hiding behind a platform-independent interface, and ensures that the interface does not expose functions that are dependent on a particular platform.

Modularity and testability Testability is the capability of the software system to enable modified software to be validated. Modularity plays an important role in testability because it supports separating concerns among the parts of the system through coupling, cohesion and the likelihood of changes, so that different parts of the system can be tested separately without being interfered by each other. Monolithic characteristic in design may result in additional efforts in testing, as error corrections in one part of the software might require retesting of the other parts or the whole system. Having to link in many different libraries also leads to increased testing effort, particularly in the case of cyclic dependencies, where unit testing and releasing become difficult and error-prone.

Modularity and domain-specific attributes Domain-specific attributes are the additional quality subcharacteristics that are required by specific domains. The relationship between modularity and domain-specific attributes depends on the particular attribute and domain context. For instance, component exchangeability in the context of service reuse [26] is one domain-specific attribute within the distributed domain, e.g. wireless computing, component-based and service-oriented applications. In this context, modularity plays an important role because encapsulation mechanism shields the business logic and implementation from the outside world and thus enables component exchangeability.

2.3 Modularity and Inter-Module Dependency

Inter-module dependency is one of many indicators and measures for achieving modularity. Excessive inter-module dependencies have long been recognized as an indicator of poor software design [37]. They diminish the ability to reason about components of the software architecture in isolation. It becomes also difficult to assess and manage change impacts.

One way to visualize these dependencies is the Design Structure Matrix (DSM)[1], which is a representation and analysis mechanism for system modeling with respect to system decomposition and integration. Several architectural styles and dependency types, e.g. cyclic and hierarchical dependencies, are detectable in this matrix. There are two main categories of DSMs: static and time-based [6]. Static DSMs represent system elements and are analyzed with clustering algorithm. Time-based DSMs represent activity flows and are analyzed with sequencing algorithms. In this paper, we focus on static DSMs to reveal software structure problems during software evolution and explore alternative architectures to improve the evolvability of the software system.

3. Research Method

We designed and conducted the dependency analysis of the control and protection system software which consists of more than one million lines of C and C++ code. The approach described in [37] was applied and we performed the following steps:

Step 1: Understand application and Dependency Structure Matrix representation.

Step 2: Create preliminary Dependency Structure Model of the application, using the hierarchical structure of the code's own namespace.

Step 3: Create conceptual architecture.

Step 4: Organize the Dependency Structure Model to reflect the intended conceptual architecture.

Step 5: Define design rules, specifying external library usage and application interdependencies.

Step 6: Perform dependency management during software evolution.

Two potential parser alternatives were considered, i.e. Doxygen and Microsoft Browser (BSC). Doxygen was in the end not selected for analyzing and parsing the source files. The reason is that it does not correctly resolve dependencies when the symbol names are not unique, i.e. Doxygen can mix up a local variable reference for a global variable reference if they have the same name. It also has problems with symbol names used in multiple contexts. The BSC module was

[1] http://www.dsmweb.org

instead chosen to be used as input for generating the initial dependency model. It processes source code written in both procedural and object-oriented languages (e.g., C and C++), capture indirect calls (dependencies that flow through intermediate files), run in an automated fashion and output data in a format that could be input to a DSM. The BSC module analysis is file based and supports member level expansion of the files displayed in the dependency model.

We used Lattix[2], a source code level DSM derivation tool to extract code dependencies and examined the following kinds of dependencies:

Class reference: If class A refers to class B, e.g. as in an argument in a method, then A depends on B.

Invokes: If a function in class A calls to a function or a constructor of class B, then A depends on B.

Inherits: If class A is a subclass of class B, then A depends on B.

Data member reference: If a function in class A makes reference to a data member of class B, then A depends on B.

Three persons were actively involved in and performed the analysis process – one researcher from the research center, one software architect and one key software developer from the development unit of the analyzed system. The focus of the researcher was to apply the tool and analysis approach on the analyzed software system, attain an overview of the dependency situation and identify hotspots in the architecture and implementation. The software architect and the key software developer from the development unit have provided with information through daily meetings to make the conclusions objective. They also supported with their comprehensive domain knowledge, especially during the iterative process of creating a conceptual architecture for the analyzed system, where they identified the subsystems and modules in each layer. The risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences. The dependency analysis process took approximately three weeks. The architecture hotspots and refactoring solution proposals for the evolution path of the software system were identified. These proposals were discussed with the main technical responsible persons and architects, documented and transferred further to the implementation teams. Additionally, the experiences described in section 5.1 are summaries of the opinions of the involved stakeholders from the development unit.

[2] http://www.lattix.com

4. Case Study

The power control and protection system is built up from a basic system which handles communication, I/O and services, and from application functions that are combined to define various products. Software development is performed by several different development teams from two separate business units and across different geographical locations. We focused on the basic system which is the platform for different product types, i.e. control and protection as well as combinations of these.

The main problem with the original software architecture was the existence of tight coupling among components, which has led to additional work to modify some existing functionality and add support for new functionality in various products. This problem was discussed during the architecture workshops with the stakeholders, including people from product management, software architecture team and key software development team. Thus, inventory of candidates for modularization through dependency analysis was identified as the first top priority architecture requirement. Accordingly, the main focus of our case study was to analyze the software architecture in terms of inter-module dependencies, and to achieve a precise dependency overview for supporting software evolution. We identified potential flaws in architecture, implementation violations and defined an evolution path of the software architecture. In addition, we succeeded to convince the management of the effectiveness of using dependency model to guide and support software architecture evolution.

4.1 Examples of Analysis

We performed static software analysis using DSM models based on source code dependencies to extract dependency relations. Since the complete assessment of components cannot be presented due to space limitations, we select a subset and exemplify with two examples from the case to illustrate component evolution through inter-module dependency analysis. The examples are chosen to be understandable for people outside the power technology domain, while still representative and illustrative for the many various discussions and solutions that occurred during the analysis. The identified hotspots are analyzed in terms of the following views: (i) problem description: the problem and disadvantages of the original design of the component; (ii) requirements: the new requirements that the component needs to fulfill; (iii) improvement solution: the architectural solution to design problems; and (iv) rationale and architectural consequences: the rationale for design decisions and architectural implications of the deployment of the component.

4.1.1 Example 1 - Web Server

The *Web Server* subsystem is used to monitor the process and status of devices with respect to measurements, events and alarms. It consists of three main parts: a third-party software module, web client application and the software interface between client and server applications. The web client application is a combination of static and dynamic web pages, client-side scripts and style sheets.

Problem Description. Two cyclic dependency problems exist and these dependencies need to be removed, since we cannot change anything to either the module without possibly affecting the others. Accordingly, they prevent us from developing, testing or releasing modules independently.

(1) The *Web Server* subsystem existed within the *Base* system as shown in Figure 2a). It consists of third-party software, which is intertwined with the control and protection system's product family. As a result, the code size of *Base* increases, and the *Base* is affected by the third-party software because *Base* needs to be updated and recompiled once there is any update or change of the third-party software in the *Web Server* subsystem. However, simply moving *Web Server* outside *Base* creates a problem of cyclic dependencies between *Web Server* and *Base* as shown in Figure 2b). The dependency matrix in Figure 4a) illustrates also the cyclic dependencies between *Web Server* and *Base*, i.e. the number in the first row indicates that *Base* uses *Web Server*, and vice versa as indicated by the number in the fourth row. Figure 4a) illustrates the dependencies among the components and visualizes the dependency violations, i.e. the implementation and architectural violations that are against design rules and design decisions. These violations are shown by the dependencies above the diagonal in the matrix (refer to [36, 31] for details). The numbers in the cells indicate the dependency strengths.

(2) The *Data* component encapsulated in *HMI Variant* subsystem is used by both the *HMI Variant* and the *Web Server* subsystem as shown in Figure 2a). To reduce the coupling between *Web Server* and *HMI Variant*, the *Data* component needs to be moved outside of *HMI Variant*. However, this creates another problem of cyclic dependencies between *HMI Variant* and *Data* as shown in Figure 2b). The dependency matrix in Figure 4a) illustrates also the cyclic dependencies between *Data* and *HMI Variant*, i.e. the number in the second row indicates that *Data* uses *HMI Variant*, and vice versa as indicated by the number in the third row.

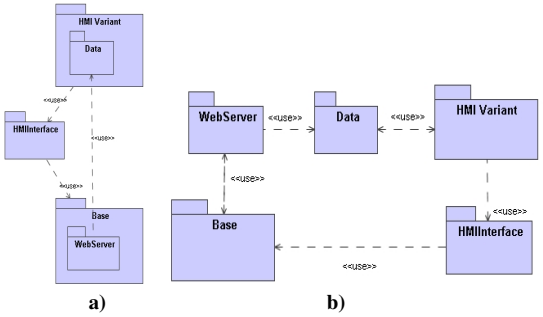

Figure 2. Conceptual view of the original correlations between Web Server and HMI components

Requirements. The *Web Server* must be isolated and moved outside *Base*. The *Data* component must be moved outside *HMI Variant*. In addition, the dependencies from *Base* to *Web Server*, as well as dependencies from *Data* to *HMI Variant* need to be removed.

Improvement Solution. The original architecture is transformed by partitioning the *HMI Variant* and *Base* respectively so that the cost for component modification is reduced. The revised conceptual architecture is illustrated in Figure 3.

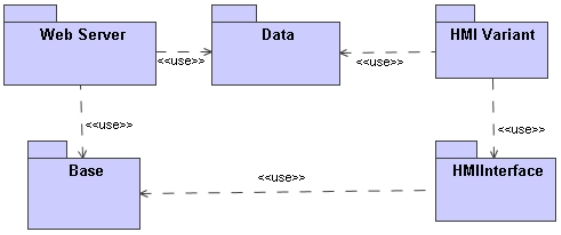

Figure 3. Conceptual view of the refactored correlations

Rationale and Architectural Consequences. The dependencies from *Web Server* to *Base* exist because some files in the *Web Server* component are used by the start-up sequence files in the *Base*. Accordingly, the implementations in the start-up sequence files were modified, and equivalent function was implemented in the application main module instead in order to remove the dependencies from *Base* to *Web Server* as illustrated in Figure 4b). In this process, we break the cyclic dependencies between *Web Server* and *Base* by moving the classes and functions that they both depend on into the application main module. The dependencies from *Data* to *HMI Variant* are caused by dead codes that are not in use any more.

The revised system architecture consists of a number of cohesive, modular subsystems and components with their implementations hidden behind well-defined interfaces. The probability that a change to one module (e.g. *HMI Variant* or *Web Server*) propagates to other modules is reduced.

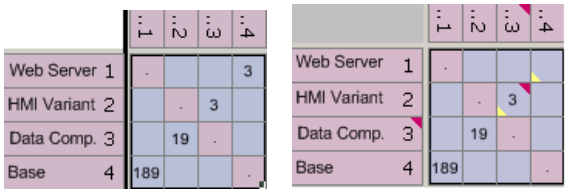

Figure 4. Dependencies before a) and after refactoring b)

4.1.2 Example 2 – Base

The *Base* software is used to provide a collection of services, as well as a platform that provides means of instantiation and configuration of application functions.

Problem Description. The *Base* software is a mixture of components that were traditionally implemented as function-oriented subsystems. They were not ordered according to any architectural styles. Direct connections and dependencies existed among components. If a change is made for a component, this implies changes to other components as well. The original coarse-grained architecture is depicted in Figure 5.

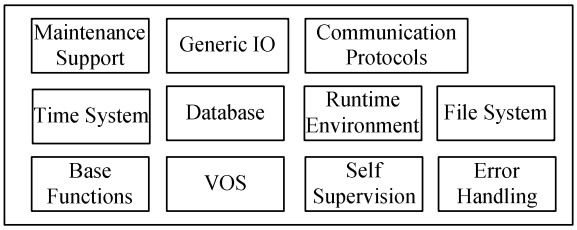

Figure 5. A conceptual view of the original software architecture

The initial DSM is created after loading the code base as in Figure 6.

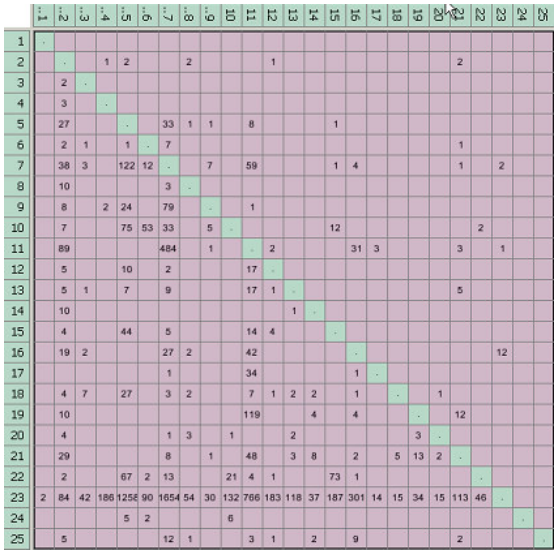

Figure 6. Initial DSM for the code base

The x-axis and the y-axis of the matrix represent the same subsystems which are numbered sequentially. The dependencies for each subsystem are read down a column. Reading column 1, we see that subsystem1 depends on subsystem23 with dependency strength of '2'. This figure reveals the tight couplings among components and violations of design decisions (shown by the dependencies above the diagonal in the matrix).

Requirements. Clear boundaries between different parts of the system need to be defined. Late source code changes should not impose ripple effects through the system.

Improvement Solution. The revised conceptual architecture is illustrated in Figure 7. It consists of three layers including Utility layer, Middle Layer and Application Layer. The conceptual architecture was attained through an iterative process, i.e. daily discussions with the software architect and key software developer, with respect to what-if scenarios (what is the impact if we change) based on the dependency information provided by the inter-module dependency model.

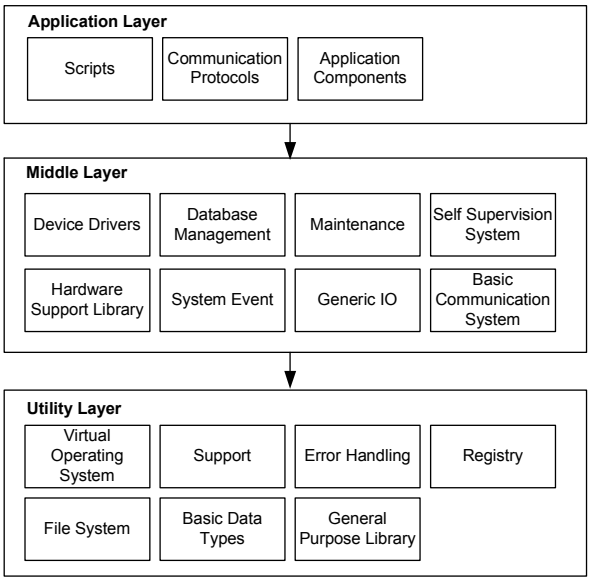

Figure 7. A conceptual architecture of the Base system

Rationale and architectural consequences. The original architecture is restructured into layered architecture, as the layers architectural pattern helps to structure applications to be decomposed into groups of subtasks at a particular level of abstraction [7]. The layered organization of software components offers a number of benefits such as reusability, changeability and portability [38]. In addition, cyclic dependencies across layers are identified as illustrated in Figure 8. For instance, reading column 6, we see that Utility layer depends on Middle layer with dependency

strength of '57', indicating architectural layering violations.

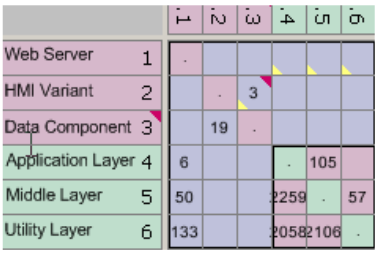

Figure 8. Dependencies after restructuring

The figure is a snapshot of the dependency model during the analysis process. The dependency violations are visualized by the dependencies above the diagonal in the matrix. As cyclic dependencies would make layers monolithic and inseparable, it is essential to break the cyclic dependencies. Two primary mechanisms [28] exist: (i) apply the dependency inversion principle; and (ii) create a new module or package, and move the classes that the cyclic dependent modules depend on into the new package.

5. Experiences and Reflections

This section presents firstly the benefits that were perceived by the involved stakeholders and secondly, our reflections through performing the inter-module dependency analysis.

5.1 Perceived Benefits of Performing Dependency Analysis Using Dependency Model

We summarize below visible benefits that were perceived and reported by the involved stakeholders in the organization.

a) It becomes easy to achieve a good overview of dependencies within the whole software system;

b) The software architects and software developers have increased potentials to do pre-studies in exploring different architectural and implementation solutions, due to the possibility of simulating changes in the dependency model without the necessity of making any modifications to the actual source code and due to the corresponding quick feedback on modifications from dependency analysis;

c) It enables a better and faster understanding of unfamiliar modules from dependency perspective; For instance, the development of *Web Server* subsystem was originally outsourced to another development unit located in another country. After the initial development, the original developers have changed their job and no one in the organization has the complete knowledge of the subsystem. However, the visualization of inter-module dependencies through the dependency model provides support for understanding the interaction of this subsystem with other parts of the system.

d) It facilitate discovery of implementation violation and perform quality check between various revisions; Design rules can be defined in the dependency model. Thus, it is possible to monitor if any implementation violations occur in the consecutive revisions to continuously check the quality of the architecture.

e) The possibility for reuse is increased; Excessive and unexpected dependencies reduce the reusability of components in different contexts and complicate the evolution of respective components, since each extension of components might affect other components. An example is managing inter-module dependencies in product line architecture. When a component is shared across multiple products, all components that this component depends on will also have to be shared or replicated in all of those products.

f) The time to do modularization work is shortened due to the quick visualization feedback from the dependency model.

5.2 Experiences and Reflections

We list below our reflections during the dependency analysis.

Gain management support Senior management generally has limited technical understanding to see the direct benefits of refactoring software architecture for improved quality, especially when there is a lack of economic models visualizing the benefits of investment. Although the software architects see the need for architecture restructuring, they usually do not have the roles of personnel resource management to execute the restructuring. In the case study, the three week dependency analysis succeeded to convince the management of the priority of architectural refactoring through the measure of dependency model. As a result, the management determined to continue with software architecture quality improvement activities instead of only focusing on providing functionalities.

Document rationale for each design decision Although the representation in the dependency structure matrix demonstrates the design decisions through the definition of design rules, e.g. the can-use and cannot-use rules, there is still a lack of explicit documentation of rationale behind the architectural decisions. Therefore, the dependency model needs to be complemented with design rationale information.

Apply routine dependency analysis as a quantitative indicator for judging the necessity of software

refactoring and for supporting the choice of design decisions The software architecture needs to evolve to accommodate changes. Meanwhile, it is also essential to define design rules and monitor if any implementation violations occur during the software evolution process. Thus, we suggest routine dependency analysis as an integral part and quantitative indicator for continuously judging the necessity of performing software refactoring. In this sense, the process is close to the idea of agile software development in terms of continuous reengineering.
In addition, the choice of any design decisions can be supported by the quantitative measures from dependency analysis. It is a challenging task to make appropriate architectural decisions especially when there is a lack of quantitative measurement of the corresponding impacts on the system. Although there exist design tactics that assist in making design decisions, their corresponding impact within a particular system is still on an intuitive and qualitative level. Therefore, we suggest complementing with dependency analysis to better support design decisions, i.e. qualitatively reason about and quantitatively measure the impacts to make more accurate estimation on workload when making architectural changes.

Combine static code analysis with dynamic information extraction The case study shows that it is beneficial to perform static dependency analysis of source code to assist in software architecture evolution. Another aspect that is of interest is to identify and analyze the runtime structure and behavior of the software, and identify the runtime components and their dependencies. An example is to reconstruct software architectures in terms of pattern recognition. Patterns whose implementation involves dynamic mechanisms will require extraction of dynamic information [17]. This suggests a combination of extracting dynamic information of a system at run time and static source code analysis.

Combine different means for improved modularization In the case study, there have been discussions about techniques and means to increase modularization, as well as the potentials of combining different approaches for improved modularization and quality attributes. For instance, studies [20, 30] have shown that aspect-oriented software development can be applied in conjunction with object-oriented programming in order to achieve better modularity, reuse and adaptability in complex software systems [31]. As part of the dependency analysis process, we have identified some means for providing modularization (as shown in Table 2) to support software evolution and to provide one way to let some part of a system change independently of all other parts. A modularization technique benefits a design only when the potential changes to the design can be well encapsulated by the technique [8]. In the case study, the improved modularization was achieved through applying several design principles, e.g. separation of concerns, encapsulation boundaries and architectural coupling reduction, together with object-oriented software engineering and layered architecture style.

Table 2. Examples of Means to Increase Modularization

Means to Increase Modularization	Examples
Design Principles	Separation of concerns
	Information hiding
	Encapsulation boundaries
	Narrow component interfaces
	Architectural coupling reduction
Software Engineering Paradigms	Object-oriented software engineering
	Component-based software engineering
	Service-oriented software engineering
	Aspect-oriented software engineering
	Feature-oriented programming
Object-oriented Design Patterns	e.g. model-view-controller
Formal Specification	Specification of interfaces between components
	Assembling of components with compatible specifications
Programming Languages	e.g. coding guidelines for enabling modularization in programming languages
Modeling Techniques	Architectural description languages, e.g. ACME
	UML being enhanced with additional modularity mechanisms and abstraction, e.g. aspects, features
Architecture Styles	e.g. layer architectural style

6. Related Work

The link between modularity and evolution was described by Simon [39] who argued that nearly-decomposable systems facilitate experimentation and problem solving. [22] examined the design evolution of one open source software product and one company software product platform through the modelling lens of design rule theory and design structure matrices. The idea of using design rules and DSM was similar to the way that we have performed in our case study. We further enrich the data with experiences and reflections through our dependency analysis of a complex industrial software system.

There exist different ways to visualize dependencies. [27] describes the concept of DSM and the application of design rules to identify violations, and to keep the code and its architecture in conformance with one another. Checking the conformance between design and implementation has been explored in [19]. Li [24] proposed object-oriented system dependency graph to calculate the impact of

changes made to a class, with focus on three relationships, i.e. containment, use/reference and inheritance. Sullivan et al. [41] and Lopes et al. [25] have presented that DSM modeling can capture Parnas' information hiding criterion [32] and is valuable for software design. [1] formalizes this reasoning by showing that modularity creates design options.

The Architecture Tradeoff Analysis Method (ATAM) [2] is a method for evaluating software architectures in terms of quality attribute requirements to achieve better architecture. It is used to expose the possible areas of risks, non-risks, sensitivity points and trade-off points in the software architecture. Since it relies on the knowledge of the architect and has no provision for code inspection, it is not a precise instrument [2] as it is possible that some risks remain undetected. As a dependency model has the feature of being able to quantitatively and thus objectively visualize the inter-module dependencies, it can be used as a complementary approach to ATAM when there is existence of code.

7. Conclusions and Future Work

In this paper, we explored the links between evolvability, modularity, as well as inter-module dependency, and described a dependency analysis of a complex industrial power control and protection system, using the inter-module dependency model. The analysis was driven by the need of improving software evolvability, and it was performed by three persons (one researcher, one software architect and one key software developer), taking approximately three weeks. The purpose of the analysis is to visualize dependencies to provide direction to hotspots in the architecture and implementation. The resulting analysis documentation was widely accepted by the stakeholders involved in the analysis process and became a blueprint for further implementation improvement. Besides, the management was convinced of the effectiveness of using dependency model as a means to guide and support software architecture evolution. Additionally, the quantitative results also convinced them of the priority of improving architecture for better quality, instead of only focusing on functionality.

Our plans are to apply dependency model in new cases and in new domains, and further complement the static analysis with dynamic execution analysis. In addition, we need to consider the impact with respect to the software system's behavior, quality and any possible tradeoffs when we introduce any modularization mechanism and technique. Thus, another research area that is of interest is to investigate the impact of the choice of modularization mechanisms, as they might have consequences for flexibility and other concerns, such as runtime qualities, e.g. performance and scalability, etc.

References

[1] Baldwin, C. Y., Clark, K. B.: Design Rules, vol 1, The Power of Modularity, MIT Press. (2000)

[2] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison- Wesley. (2003)

[3] Bennett, K., Rajlich, V.: Software Maintenance and Evolution: a Roadmap. The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press. (2000)

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

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

[6] Browning, T. R.: Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions, IEEE Transactions on Engineering Management. (2001)

[7] Buschmann, F. et al.: Pattern-Oriented Software Architecture: A System of Patterns. Chichester, NY: Wiley. (1996)

[8] Cai, Y., Huynh, S.: An Evolution Model for Software Modularity Assessment. Fifth International Workshop on Software Quality. (2007)

[9] Chapin, N. et al.: Types of Software Evolution and Software Maintenance, Journal of Software Maintenance and Evolution: Research and Practice. (2001)

[10] Clements, P., Bachmann, F., Bass, L. et al.: Documenting Software Architectures – Views and Beyond. (2007)

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

[12] Developing Architecture Views. http://www.opengroup.org/architecture/togaf8-doc/arch/chap31.html. (visited 2008)

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

[14] Feng, T., Zhang, J., Li, W.: Applying Change Impact Analysis and Design Metrics in CBR Based Software Design Improvement, Proc. of ISCIT. (2005)

[15] Fitzpatrick, R. et al.: Software Quality Challenges. 26th International Conference on Software Engineering. (2004)

[16] Grady, R., Caswell, D.: Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs, NJ, PrenticeHall. (1987)

[17] Guo, G. Y., Atlee, J. M., Kazman, R.: A Software Architecture Reconstruction Method. WICSA. (1999)

[18] Huynh, S., Cai, Y.: An Evolutionary Approach to Software Modularity Analysis, 1st International Workshop on Assessment of Contemporary Modularization Techniques. (2007)

[19] Huynh, S., Cai, Y. et al.: Automatic Modularity Conformance Checking. ICSE (2008)

[20] Improve modularity with aspect-oriented programming. http://www.ibm.com/developerworks/java/library/j-aspectj/. (visited 2008)

[21] ISO/IEC 9126-1. International Standard. Software Engineering – Product Quality – Part 1: Quality Model. (2001)
[22] LaMantia, M. J., Cai, Y. et al.: Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases. WICSA. (2008)
[23] Lehman, M.: Laws of Software Evolution Revisited. Software Process Technology, 5th European Workshop EWSPT. (1996)
[24] Li, L.: Change Impact Analysis for Object-Oriented Software, PhD thesis, George Mason University, Virginia, USA. (1998)
[25] Lopes, C. V., Bajracharya, S. K.: An Analysis of Modularity in Aspect Oriented Design, Proc. of AOSD. (2005)
[26] Lüer, C. et al.: The Evolution of Software Evolvability. IWPSE. (2001)
[27] MacCormack, A., Rusnak, J., Baldwin. C. Y.: The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. HSB Working Knowledge. (2008)
[28]Martin,R.: Acyclic Dependency Principle - Granularity.http://www.objectmentor.com/resources/articles/granularity.pdf (visited 2008)
[29] McCall, J. A., Richards, P. K., Walters, G. F.: Factors in Software Quality. National Technical Information Service. (1977)
[30] Mens, T., Demeyer, S.: Software Evolution. Springer. (2008)
[31] Padayachee, A., Eloff, J.H.P.: The Next Challenge: Aspect-oriented Programming. Proc. of the Sixth IASTED International Conference on Modelling, Simulation, and Optimization. (2006)
[32] Parnas, D. L.: On the Criteria to be Used in Decomposing Systems into Modules. (1972)
[33] Parnas, D. L.: Designing Software for Ease of Extension and Contraction, IEEE Transactions on Software Engineering. (1979)
[34] Pei Breivold, H., Crnkovic, I., Eriksson, P.: Analyzing Software Evolvability. Proc. of COMPSAC. (2008)
[35] Rowe, D., Leaney, J.: Defining Systems Evolvability – a Taxonomy of Change. Proc. of the IEEE Conference on Computer Based Systems. (1998)
[36] Sangal, N.: Expressing Software Architecture with Inter-module Dependencies. EclipseZone. http://www.eclipsezone.com/articles/lattix-dsm/. (visited 2008)
[37] Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture, OOPSLA. (2005)
[38] Sarkar, S., Rama,: A Method for Detecting and Measuring Architectural Layering Violations in Source Code. (2006)
[39] Simon, Herbert A.: The Architecture of Complexity. Proc. of the American Philosophical Society 106: 467-482, repinted in idem. (1981) The Sciences of the Artificial, 2nd ed. MIT Press, Cambridge, MA, 193-229. (1962)
[40] Stoermer, C., O'Brien, L., Verhoef, C.: Moving Towards Quality Attribute Driven Software Architecture Reconstruction, Proc. of the 10th Working Conference on Reverse Engineering. (2003)
[41] Sullivan, K., Cai, Y., Hallen, B., Griswold, W. G.: The Structure and Value of Modularity in Software Design, SIGSOFT Software Engineering Notes. (2001)