IT University of Copenhagen

Re-engineering for Evolvability

Considering social as well as technical requirements for software products

Hataichanok Unphon

Thesis submitted for the Degree of Doctor of Philosophy

16 October 2009

ABSTRACT

Today's software products are used by a number of organisations to serve their businesses. The software is often customised to support a variety of needs and uses in an organisation. Over time, software has evolved to support changes in the way it is used. The challenge for the software to maintain its *Evolvability*—that is, its adaptability and at the same time its sustainability with respect to the way it is used and further developed—deserves serious consideration in academia and industry.

The goal of this research is to propose engineering discipline for enhancing continuous evolvability. The research is designed as a qualitative empirical study using two case studies—DHI Water Environment and Health (DHI) and EASI-WAL, an open source project let by a Belgian government agency—that re-engineer their software products using software product line approach, and an interview study with different product developing companies.

The study with DHI took place over a time span of 2.5 years as an action research project. The study with EASI-WAL triangulates the first one; the research took place during a three-month stay in Belgium. The interview study was performed as yet another triangulation in order to see whether the observed practices in the two cases studies can be observed in other organisations as well.

The two case studies (a long-term involvement and a short-term involvement case study) centre on the re-engineering of socially embedded systems in which the design and evolution depend on usage, development cooperation, and changes to the technical base. Based on the initial empirical research, an *evolvability framework* is identified that contains six contextual dimensions (business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection) which need to be considered when evolving the software product in a sustainable manner. The research highlights architectural changes and changes to the architectural practice when introducing a product line approach. E.g. to promote awareness of architecture in everyday development, the *build hierarchy* in DHI was changed to represent the design architecture and thus promote the alignment of design and code architecture.

As a result, this thesis proposes and evaluates an *Architecture-Level Evolvability Assessment (ALEA) method* allowing the development team to assess adaptability as well as sustainability with respect to the six contextual dimensions when evolving the software product.

The interview study investigates the architecture practices of product developing companies. Based on interviews with architects and in some cases also team members of 13 software product teams in eight organisations, the study confirms that architecture is communicated and maintained through socialisation rather than codification in form of documentation. Architecturing is carried out by *walking architecture,* a lead developer or chief architect cooperating and communicating with the developers. Based on the analysis, the study discusses the *interaction of tools and representations and social protocols* to promote *architecture awareness* in product evolution.

The results indicate the importance of considering both the social and the technical dimension when improving architecture and software development.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude and grateful acknowledgement to my supervisor, Associate Professor Dr. Yvonne Dittrich for her advice, enthusiastic encouragement, and patient throughout my study. I also acknowledge Professor Dr. Peter Sestoft, Professor Søren Lauesen and Associate Professor Kasper Østerbye for their comments and recommendations.

I would like to express my sincere gratitude to the Danish government and DHI Water Environment and Health (DHI) for partial financial support through the Evolvable Software Products (ESP) project. I am most grateful to IT University of Copenhagen for supporting my studentship. I should like to extend my grateful thanks to Professor Dr. Patrick Heymans from the Faculty of Computer Science, University of Namur, Belgium and Dr. Muhammad Ali Babar from LERO, University of Limerick, Ireland for hosting my three month stay abroad. I should like to extend my gratitude to Dr. Wolf-Gideon Bleek for his constructive feedback and help in setting up the interview study.

I record my thanks to all DHI staff members for the continuous help and valuable guidance. I kindly thank to all my interviewees in Belgium, China, Denmark, Germany, and Switzerland for taking part in my interviews that enabled me to fulfil my empirical works.

Finally, grateful acknowledgement is extended to my parents, Samor and Kannikar Unphon, my boyfriend, Christian Edinger, and my friends for their dedication, support and constant encouragement throughout my PhD education.

CONTENTS

PART I

1

INTRODUCTION

Today's software products are often customised to support a variety of needs and uses in an organisation. Over time, the software products evolve to support changes in the way it is used. The challenge for the software is to stay adaptable in order to serve the needs of business and at the same time develop in a manner that assure its sustainability with respect to its usage and future development. This is what we call *evolvability*. The research presented in this thesis aims to establish an engineering discipline for evolving software products into more flexible designs, to enhance the evolution process in daily practice, and to prolong the software's productive life. Note that the notion of engineering discipline refers to tools, practices and methods that are accountable, repeatable, and if possible it can be generalised.

This thesis is grounded upon empirical research (two case studies and an interview study), which is motivated by the industrial demand to carry out evolution more efficiently. The empirical research enriches the understanding of the concept of evolvability from an industry perspective. Due to the aim of this thesis of proposing an engineering discipline, an action research approach is carried out. The approach is based on empirical understanding of the situation before introducing a new element into the situation. The research focuses on software architecture. The major part of this empirical research elaborates evolvability from re-engineering using a software product line approach [21]. The distinction between this research and prior research of software product lines is the focus on the type of software product. Prior research mostly focuses on *technically embedded systems* in which design decisions of the systems are constrained by interfaces to hardware and mechanical specification. Typical examples are drivers for printers, mobile phones, and security alarms. In contrast to this research, the focus is on *socially embedded systems* in which design decisions of the systems underline the importance of human interaction and cooperation via the systems for social activities. Examples here include enterprise resource planning (ERP) systems, hydraulic simulation software and e-Government applications. Note that the second example is the main case study presented in this thesis and the last example is a supplementary case study.

In the beginning of the research, I, as a researcher, only looked at technical contexts. For example, I followed the formal approach of a product line architecture that was oriented to serve technical requirements, not the social requirements. However, working in the field and understanding contingencies of software development practice, it became clear that not only the technical contexts, but also the social contexts (i.e., use, business, development organisation and practice) play an important role. Floyd et al. [29] have already emphasised bringing the social contexts along with the technical in the essence of software development. However, they did not explicitly explain how to do that with respect to software architecture practice.

The research presented in this thesis points out the importance of combining both technical and social aspects to a framework for evolvability. This framework further supports the architecture assessment method, so-called Architecture-Level Evolvability Assessment (ALEA), one of the proposed engineering disciplines in this thesis. ALEA emphasises both technical and social requirements as inputs for architecting, which contributes to the answer of "*how to embed architecture work in everyday software development practice, and raising architecture awareness.*" The research proposes engineering disciplines for making use of architecture throughout the software life cycle. Through interaction with industry, we have proposed engineering discipline does support software evolvability.

1. Evolution and re-engineering of software products

The notion of evolution appeared in Lehman's work since the 1980s [11, 39, 40] in the criticism of the term software maintenance. Due to the software product' changing requirements, the word "maintenance" should be replaced by "re-engineering" or "evolution" [65], which represents much broader activities. These activities not only fix faults in the original implementation, but also add new functionalities, dramatic enhancements, and alter original design functions into a new form.

The success of many businesses is critically dependent on software products. Businesses need to be increasingly flexible and responsive to the marketplace, and to develop and market new products and services in a timely manner. To do this, software needs to be as flexible as possible and accommodating for rapid modification and enhancement. Handling software evolution properly is vital to the success of any company.

Software needs to be upgraded with major enhancements done within a short time frame in order to meet new business opportunities and reduce the "time to market" for new products and services. After many changes, the evolvability of software often dramatically decreases. Many software products become difficult to understand and change. Apart from that, software often has been originally optimised for performance or space utilisation at the expense of comprehension. Sometimes, the initial program structure has been corrupted by a series of changes over time which leads to dead-end evolution. To simplify these problems, a company may decide to re-engineer the software to improve its structure and understandability [57] as well as business value, market share and development practice.

The critical distinction of re-engineering from new software development is that the old software product acts as a specification. More precisely, the functionality of the software does not change. The costs of re-engineering depend on the extent of the work that needs to be carried out. For example, major architectural changes incur high additional costs. Using modern software engineering methods do not guarantee that the new development will not jeopardise the software product's evolution. This raises the question of "*how to develop software for long term evolution*", which leads to the research described in this thesis "*re-engineering for evolvability*".

2. Research questions, approach and scope

This thesis attempts to answer research questions (*RQs*) on evolving software products by discussing the following issues:

RQ1: What is software evolvability?
RQ2: Do software developing companies manage to maintain evolvability of their software products over a long lifetime, and how do they do that?
RQ3: How to develop software products for evolvability?

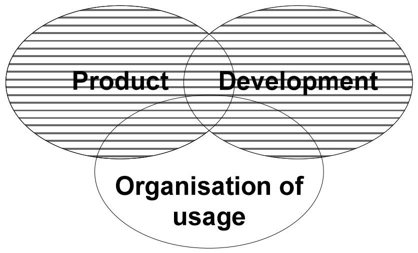


Figure 1. The research scope

In order to focus attention on *RQ1* and *RQ2*, as well as deliver the engineering discipline posted in *RQ3*, an empirical qualitative research approach is selected. The mainline of research is a case study following the cooperative method development (CMD) approach [28]. The complemented case study addresses the design for product line architecture and making use of the architecture beyond the design phase. The research is expected delivering an engineering discipline for practitioners to work with evolvability. CMD allows combining empirical research with the introduction of an engineering discipline and its evaluation. Besides that, an interview study following a grounded theory approach is conducted. Results from the interview study triangulate [41] the mainline of the research as well as contribute to the answers of the above research questions. The answers are expected to cover the scope of the research, as shown in Figure 1, which includes product and development, partly in relation to the organisation of usage that is the organisation using the software and how the software use is organised.

3. Thesis outline

To summarise, this thesis points out the importance of social contexts as well as technical contexts when evolving software products. The research is an empirical study that is motivated by the industrial demand to carry out evolution in a systematic way. The research scope presented in Figure 1 draws the attention to product development and partially to organisation of usage. The result of this research is an engineering discipline for handling evolvability that embeds architecture work in everyday software development practice and raising architecture awareness.

This thesis is divided into two parts; Part I: research introduction (Chapter 1-5); and Part II: a series of articles that contributes to the research (Chapter 6-11). Chapter 1 explains research topics and problem formulations for this research. Chapter 2 places the research in relation to different research communities. Chapter 3 explains research approaches including evaluation and credibility for this research. Chapter 4 describes how the articles that comprise the main body of the text will address the topic. Chapter 5 provides a general conclusion which integrates the material addressed in Chapter 6-11.

RELATED RESEARCH

Re-engineering for evolvability brings together three existing topics: software architecture, evolvability, and software product lines. This chapter provides an overview of how the three existing topics relate to this research.

1. Software architecture

In programming, the term *architecture* has been used since the late 1960's [16] for discussing the high-level design of software products [55]. In the early 1970s, Parnas [45, 46, 47, 48] came up with many of the fundamental tenets and principles in software architecture. Based on Bass et al. book [7, p. 45], "*Today, architecture as a field of study is large and growing because it left the realm of deep thinkers and visionaries and made the transition into practice.*" Some architectural ideas have been refined and applied to such an extent that it has become an accepted state-of-the-art practical approach to software engineering, which is the main focus of this research. This section presents concepts and roles of software architecture, as shown in Sub-section 1.1 and 1.2 respectively.

1.1. Architecture defined

To date there is no generally agreed upon definition of software architecture. The definitions provided here cover a number of different concepts, with key characteristics highlighted by the author in bold letters.

"*Software architecture is the structure of the* ***components*** *of a program/system and their* ***interrelationships****, along with principles and guidelines governing their design and evolution over time.*" [30].

"*An architecture is the* ***set of significant decisions*** *about the organization of a software system, the selection of* ***structural elements,*** *and their interfaces by which the system is composed, together with their* ***behaviour*** *as specified in the collaborations among those elements, the composition of those elements into progressively larger subsystems, and the* ***architectural style*** *that guides their organization -- those elements and their interfaces, their collaborations, and their composition.*" [37].

The IEEE Recommended Practice for Architectural Description of Software Intensive Systems [56], referred to as IEEE 1471-2000, defines "architecture" as "*the fundamental* ***organisation*** *of a* ***system*** *embodied in its* ***components****, their* ***relationships*** *to each other and to the* ***environment*** *and the principles guiding its design and evolution.*"

Apart from that, the following definitions are samples from software architects and software engineers given at the interview study for this research. They reflect the current understanding of software architecture among practitioners of this discipline. Again, the key characteristics are highlighted in bold letters by the author.

Software architecture is:

"*The* ***segmentation*** *of the software which is segmented to module-based architecture…*"

"*The way we are* ***assembling*** *the large building blocks. ... some kind of* ***high-level*** *patterns, for example, MVC.*"

"*How you take different* ***components*** *and put them together, how you handle* ***dependencies*** *(between components) ...*"

"*An* ***overall*** *communication of the software.*"

"***Design of the code***."

"*... A* ***blueprint*** *of how you organise software applications, typically in a class diagram or some kind of UML diagram.*"

"*… mainly providing an* ***overview*** *for development teams, to get an overview ...* "

"*Breaking (systems) into large* ***pieces****. ... It could be a huge piece, or all the way down ... to try to lay down object-oriented.*"

"***Skeleton*** *of software, how it is* ***structured****.*"

"***Stack*** *of technology.*"

Among many definitions for software architecture, the definition used in this thesis is taken from [7], the most cited book in the software architecture topic[1], "*Software architecture is the structure or structures of the* ***system****, which comprise software* ***elements****, the externally visible properties of these elements and* ***relationships*** *among them.*"

1.2. Architecture used

Software architecture provides a unifying theme for engineering and management of compromises that must be made to complete a product [58]. The pragmatic role of software architecture in software development activities addressed in this research are (*i*) representing a set of design decisions, (*ii*) communication and knowledge transfer, (*iii*) performing quality attributes assessment, (*iv*) managing evolution and (*v*) introducing product lines. The first three roles are explained below while the last two are addressed in later sections.

In a software lifecycle, the design of software architecture is located early in the lifecycle. The design is the first step after the specification requirements and is followed by later phases such as detailed design and implementation. The requirements tell *what* the software should be able to do and the architecture describes *how* that should be achieved. The implementation following the architecture will be divided into the prescribed elements that interact with each other in the prescribed fashion. Each element must satisfy its role in relation to the others as dictated by the architecture.

When and how the software is developed depends on the software life cycle (e.g., iterative or evolutionary). In the traditional waterfall development model [52] or the contract-first development, stakeholders design the complete architecture before the implementation takes place. In contrast to agile development [23] or some of the in-house product

[1] Cited by 3,168 articles according to http://scholar.google.com, last visited 27 July 2009.

development practices, the stakeholders continuously design the architecture throughout the lifecycle of the project. In different approaches, different emphasis is put on architectural knowledge handling. The waterfall model emphasises creating documentation, e.g. design documents, whereas the agile development stresses face-to-face communication. The latter emphasis will be elaborated on in Chapter 8 and 11.

The software architecture addressed in this research can be seen as an implicit theory of how the structure of the program and its parts addresses the concerns of the usage [43] or explicitly represented in an architecture document. No matter what the architecture represents, stakeholders must be able to use the architecture as a basis for mutual understanding, communication and negotiation. For example, a chief architect may use the architecture as an introduction to the software for new project members. Architecture is often mentioned intensively during the design phase of software development.

Software architecture plays a significant role in shaping the software's quality attributes. Architecture assessment is considered an effective mechanism for identifying potential architectural risks and questionable design decisions early in the software's development life cycle. The aims of architecture assessment are to evaluate the architecture's ability to deliver a system capable of fulfilling the quality requirements, and to identify potential risks. Contemporary software architecture research increasingly emphasises the importance of having a defined review process. As a consequence, the architecture review research community has developed several dedicated methods to support the review process which is becoming relatively mature and rich, e.g. Scenario-based Architecture Analysis Method (SAAM) [36], Architecture Trade-off Analysis Method (ATAM) [34], Architecture Reviews for Intermediate Designs (ARID) [22], and Architecture-Level Maintainability Analysis (ALMA) [12]. However, a survey of the state of practice in software architecture reviewing [6] indicated that the usage of those proposed methods was relatively limited. If the proposed methods are used, they are rarely used "out of the box". In other words, most architecture reviews occur on an ad-hoc basis.

Besides the structure of the software being developed, other contexts also influence the architecture, e.g. organisation of software engineering, or work practice. An example of the organisation influencing architectural design decisions will be discussed in Chapter

6. An example of the work practice influencing the decisions will be elaborated on in Chapter 7.

From my point of view, when people talk about architecture or architecture assessment, rather than embedding it in the development practice, they put it beside. Even though software architecture research is often validated empirically [19], there is a lack of empirical research of the "anchoring" of architecture in developer's work practices, businesses and use. Schougaard et al. [53] conducted an empirical study observing architectural techniques used in successful companies and analysing mismatches to the software architecture research. One of their findings was that when architects and developers use the techniques related to architectural quality, they tend to focus less on the architecture in relation to business and users. In support of this finding, the research discussed in this thesis aims at proposing architectural tools, practices, and methods that emphasise the importance of business and use contexts as well as technical contexts. Chapter 8 and 10 are examples of how to embed and establish a systematic approach for making use of architecture and architecture assessment in the development practice. The proposed architecture assessment covers necessary contexts for stakeholders to improve the capability to better understand and analyse systematically the impact of stimuli to the architecture.

2. Evolvability

This section reviews evolvability in literature and how it can be achieved in industry, as shown in Sub-sections 2.1 and 2.2, respectively.

2.1. Software evolution and evolvability

Belady and Lehman [11] first introduced the term "evolution of the software" and used that term to describe the sequence of changes to a software system over its lifetime which encompassed both development and maintenance. They made a number of observations about the size and complex growth relating to 21 releases of the OS/360 operating systems software. In the mid seventies, they proposed their laws of software evolution. Programs embedded in human activity, so called *E*-type programs, create changes in the use context, which in turn create

Table 1. Lehman Laws of Evolution

No.	Brief name	Law
I	Continuing Change	An *E*-type program that is used must be continually adapted or else it becomes progressively less satisfactory.
II	Increasing Complexity	As a program is evolved, its complexity increases unless work is done to maintain or reduce it.
III	Self Regulation	The program evolution process is self-regulating with close to normal distribution of measures of product and process attributes.
IV	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate on an evolving system is invariant over the product lifetime.
V	Conservation of Familiarity	During the active life of an evolving program, the content of successive releases is statistically invariant.
VI	Continuing Growth	Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
VII	Declining Quality	*E*-type programs will be perceived of as declining quality unless rigorously maintained and adapted to a changing operational environment.
VIII	Feedback System	*E*-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved

change requirements for the software. Lehman [40] further elaborated on the laws in the context of real systems using a small number of later releases of a general-purpose batch operating system. The initial laws have been periodically revised [38] as increasing insight and understanding have been achieved. The set of revised laws are given in the Table 1. These laws have been widely recognised and accepted.

Following this criticism, the traditional term maintainability was replaced by the term evolvability. For example, Cook et al. [24] developed evolvability on top of the ISO 9126 maintainability characteristics and proposed the evolvability measurement at different levels of abstraction, i.e., at pre-design, architectural, detailed design, and source code levels. They have further proposed that the concept of evolvability brings together factors from three main areas: (*i*) software product quality, (*ii*) software evolution processes, and (*iii*) the organizational environment in which the software is used. Breivold et al. [15] proposed an evolvability model as a framework for analysis of software evolvability. Their proposed model is a union of quality characteristics with respect to changes and evolution of a software intensive system. The subcategories, i.e., analysability, integrity, changeability, extensibility, portability, testability and domain-specific attributes, serve as a checkpoint for evaluation. Independent of aforementioned research, Unphon et al. [61, 62] proposed an evolvability framework based on empirical research that not only looks at technical dimensions, but also social dimensions. The evolvability framework comprises six different contextual dimensions: business context, use context, software engineering organisation, software engineering practice, technical infrastructure and technical selection, and serves as a guiding framework for architecture evaluation [63]. Many prior researchers [13, 20, 24, 49, 51] included functional requirements from the use contexts for the evaluation, but they do not investigate how the new design impacts the other contexts (e.g. business), nor how the changes in the design are related to organisation and work practice in software development. Further details of this framework and evaluation will be found in Chapter 6-9 and 11.

2.2. Designing evolvable software products

Borches and Bonnema [13] have reviewed the benefits of designing evolvable software products that are considered to be the best practice in many industry domains. For example, companies can benefit from a software product that can adapt to changing requirements or different environments at a cost lower than what is needed to build a new system. Evolvability enables easier insertion of new technology and mitigation of the risk of obsolescence of the products. In addition, evolvability affords additional flexibility, as the company can either reuse the existing infrastructure to tackle changing requirements or develop a new product.

By designing for evolvability, software products will be better suited to cope with unknown future requirements. Architecture has been known as a crucial factor to address evolvability [51], meaning that, to be capable of accommodating change, architecture must be specifically designed for it. However, modifying an architecture can have widespread effects on both the product's functionality and its performance. Therefore, stakeholders have to be aware that the architecture will have to face unpredicted situations, and try to minimise the effects when those situations occur. This is usually delegated to the designer's intuition. Borches and Bonnema [13] have proposed Design for Evolvability (DfE) to steer a design process in a direction where evolvability is designed into a system. DfE focuses on the creation of an evolvable architecture, the description of an evolutionary development process, and establishing an evolutionary environment. However, the research presented in this thesis takes its own existing evolvability framework (Chapter 7) one step further. This framework is embedded in the engineering discipline for designing evolvable software products (Chapter 10). Based on the research discussed in this thesis, the notion of evolvability is redefined as technical adaptability and sustainability with respect to organisational or business, as well as technical dimensions.

3. Software product lines

Software product lines (SPL) have been recognised as an approach for product developing companies to increase their productivity and flexibility for changing requirements. SPL are defined as "*a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [21]. Software product line engineering (SPLE) is a software engineering paradigm that institutionalises reuse throughout the software development process. SPLE dedicates a specific process, named Domain Engineering, to the development of reusable artefacts, a.k.a. core assets. These core assets are then reused extensively during the development of the final products, which is Application Engineering. The final products in a software product line share much of their software architecture and implementation, often because every system is derived from the same core assets. The true benefit of sharing software architectures (i.e., reuse) comes when they are applied in the form of product line architectures [14]. Product line architecture (PLA), a.k.a. reference architecture [64], is "*a single specification capturing the overall architectures of a series of closely related products*" [42].

The main challenges in designing PLA are based on the fact that (*i*) the architecture has to deal with many different products and releases at the same time and (*ii*) each product has many stakeholders involved, especially in large companies [14, 60, 64]. In terms of evolution, all main assets of software product lines evolve constantly because the requirements on the products evolve. The requirements can be initiated by existing products or by new products that need to be incorporated in the software product line. The changing requirements affect the requirements for entire product line, thus causing the product line architecture to evolve.

Change requirements are often triggered by use. Many product developing companies do use their own software products to serve their business. These companies also feed their new requirements into the development process just as their customers do. It is widely recognised that the organisational structure of companies influence the success of the execution of the process surrounding software product line development. Prior study in software product lines exclusively focus on the organisation of development. Sub-section 3.1 and 3.2 will elaborate this point in detail. However, the companies also need to focus on the organisation for usage and the collaboration between organisation of usage and development. The research discussed in this thesis underlines this need.

3.1. Organisational structure for product line engineering

One of the principal characteristics of a successful software product is that, over time, it will be reused and adapted for purposes that become increasingly different from the product's original purpose [9]. Having effective evolvability is not simply a matter of having the right software and system architecture. Organisational structure, management practices, and personal support are also affected. Brownsword and Clements [17] report the experiences of CelsiusTech Systems AB of Sweden., a company that builds large, complex, embedded, real-time shipboard command-and-control systems as a product line, developed in common from a base set of core software and organizational assets. The report describes the changes that CelsiusTech had to make to its software, organizational, and process structures to redirect the company towards a product line approach that yielded substantial economic and marketplace benefit to the company.

Dager [26] reveals the importance of organizational structure issues based on the software product line experience at Cummins Engine Inc., the world's largest manufacturer of large commercial diesel engines. The company changed its IT strategy to have a separate core asset group providing core assets to the product-building groups. Even the development process is quite different from that of other companies, but the organization plays a role in the development of a successful software product line.

Bosch [14] has discussed a number of organisational models that can be applied when adopting a software product line approach to software development, i.e., development department, business units, domain engineering unit, and hierarchical domain engineering units. For each model, he describes the situations in which it is most applicable, as well as describing the advantages and disadvantages of the model and providing an example of a company that employs the model. These models have been further observed and identified into three basic structures: product-oriented, process-oriented, and matrix organisations [64]. In any case, management is necessary for the orchestration of the entire product line effort [21]. An example of the effort will be mentioned in Chapter 11.

3.2. Integrating agile software development and software product line approaches

Agile software development methods, e.g. extreme programming (XP) [10], offer an answer to the eager business community asking for cheaper, along with faster and more flexible software development processes [4]. In agile development practice, developers concentrate only on the functions needed immediately, delivering them fast, collecting feedback, and are thus able to react rapidly to business and technology changes [23]. Tian and Cooper [59] have compared agile software development paradigms and software product line development paradigms, and found that both are being promoted as means to reduce time to market, increase productivity, improve quality, and gain cost-effectiveness and efficiency. Integrating one paradigm into another is, however, associated with a unique set of challenges related to the underlying philosophies on which each paradigm is based [5, 18, 32, 59]. In contrast to the software product line approach, which requires upfront design to set up the architecture for product families, agile methods propose a simple, incremental design that merely designs for the product at hand [18]. The software product line approach addresses longer-term strategic objectives related to life-cycle product management, whereas agile software development addresses short-term tactical objectives, such as single projects developing one specific product [32]. The differences between the software product line approach and agile software development are also noticeable in the organisation of development team. Based on [59], software product line organisational management will be more likely to move their best people to core asset development and assign product development tasks to developers with average skills. In agile software development, all team members work together on the product development.

Recently, Ali Babar et al. [5] have presented an empirical study of the convergence of software product lines and agile software development practices, as well as the leverage to improve agile software development through product line architecture. They promoted architecture documentation, e.g. design decision documentation, for the use of product line architecture, whereas the research presented in this thesis emphasises the use of architecture as integration of architecture and architecturing in the development practice. This focus is not only specific with respect to problem of an architecture approach but provides a complete perspective to all aforementioned research. Further detail will be presented in Chapter 9.

To sum up, software architecture is very much focusing on technical design. Although the use context has come into the design as functional and non-functional requirements, the impact of changes on the use, business, and organisation is rarely discussed or taken into account. Though the synergy of social and technical contexts in evolving software products has been discussed, the question of how to achieve that in practice is still open.

3

RESEARCH APPROACHES

This thesis is conducted as qualitative empirical research comprising two case studies (i.e., long-term/main case study and short-term/supplementary case study), and an interview study. The main reason of choosing the qualitative empirical research is that this thesis questions the "*why and how*", not just the "*what*", of *evolvability*. The long-term case study offers an opportunity for in-depth understanding of, and the reasons that govern, evolvability. Moreover, the research aims to provide engineering discipline for evolvability. Thus, the research method must provide the leverage to introduce and evaluate the engineering discipline in the long-term case study, as well. To date, cooperative method development (CMD) [28] is the most structured method that provides this leverage. However, to deal with the trustworthiness issue, the long-term case study is complemented by the short-term case study and the interview study. Especially the interview does not only confirm the results of the case studies, but also provides additional insights.

This chapter is outlined as follows: Section 1 elaborates on research methods; Section 2 is the description of two case studies and an interview study; Section 3 summarise evaluation and creditability.

1. Research methods

In order to get in-depth understanding of evolvability, as well as introduce and evaluate engineering discipline, cooperative method development (CMD) is the main research starting applied throughout the research discussed in this thesis, along with interviews and grounded theory. Sub-section 1.1 elaborates on CMD in detail. Sub-section 1.2 presents the interviews and grounded theory.

1.1. Cooperative method development (CMD)

Cooperative method development (CMD) is a domain-specific adaptation of action research aiming at understanding and improving industrial practitioners' practice [28], and at the same time improving method processes and tools. CMD is implemented as evolutionary cycles in which research is inspired by the industrial practitioners' perspective. Each cycle has three phases: (a) understanding practice, (b) designing improvements, and (c) implementing and observing improvements. Each phase is elaborated on as follows:

(a) Understanding practice starts with qualitative empirical investigations into the problem domain. This phase aims at understanding and explaining practitioners' existing practices based on their historical and situational context in order to identify challenges.

(b) Designing improvements brings the challenges from the previous phase and proposes possible solutions.

(c) Implementing and observing improvements is to apply the solutions from the previous phase and follow the consequences of the changes. Researchers and practitioners will evaluate and summarise the consequences together. Consequently, the researchers will build the base for the scientific evaluation of the proposed improvement measures.

In this thesis, the CMD is applied for two case studies: the long-term/main case study and the short-term/supplementary case study. Section 2 gives an example of how the main case study carries out the CMD. In Chapter 6-8, 10 and 11, the CMD will be mentioned again in the research approach/method section.

1.2. Interviews and grounded theory

Interviewing as a research method [50] involves the researcher asking questions and receiving answers from interviewees. Interviews can be structured, semi-structured, or unstructured. Structured interviews have predetermined questions with fixed wording, usually in a pre-set order. An example of a structured interview is a survey. Semi-structured interviews have predetermined questions, but the order can be modified or omitted based upon the interviewer's perceptions. Unstructured interviews use conversation to develop a common area of interest and concern between interviewer and interviewee. In this thesis, unstructured interviews are performed in order to find out "*how do product developing companies manage to maintain evolvability of their products over a long lifetime?*"

A strategy for collecting and analysing the interviews used in this thesis is the grounded theory [25]. The grounded theory approach was derived from a combination of Chicago style Interactionism and Pragmatism [31] in terms of data collection and analysis. In this thesis, data collection and analysis are an iterative process of switching between interviews and developing a concept. To build up the concept, different levels of coding from the interviews are combined in a reflexive manner. The grounded theory method and the results of the interviews will be detailed in Chapter 9.

2. Two case studies and one interview study

This section describes two case studies: a long-term/main case study at DHI Water Environment Health and a short-term/supplementary case study at EASI-WAL, as shown in Sub-section 2.1 and 2.2 respectively. The interview study is presented in Sub-section 2.3.

2.1. DHI case description

DHI Water Environment Health (DHI) is a pioneering organisation that develops software applications for hydraulic modelling. In 1972, System 11 and System 21 were two of the first computational modelling systems developed at DHI to simulate water flow patterns with the help of one-dimensional and two-dimensional models. A three-dimensional simulation was developed in the 1980s. Originally, the organisation focused on hydraulic characteristics research, not on software engineering. Software development and software maintenance were challenged only on a small scale.

In the late 1980s, DHI released the MIKE 11 and the MOUSE software products. Both products originated from System 11 following requests for different usages, i.e., open channels and pipe networks. MIKE 11 and MOUSE are stand-alone Windows-based applications. The main users of these products are hydraulic and environmental consultants who perform simulations of hydraulic conditions (e.g., water level and flow), and analyse the hydrological effects of environmental change. Due to varying market needs, ownership was split into different consultancy departments; and during the past three decades MIKE 11 and MOUSE have been developed and maintained in parallel. Released in 2005, MIKE URBAN followed requests to have a more complete and integrated modelling framework for both water supply and wastewater systems.

After decades of successful use and development, the requirements of the software have evolved as well. In particular, there is a growing tendency that the software be used in a more general setting, e.g., scheduled forecasts. The company was faced with the challenge of identifying and developing a kernel for data handling, simulation setup, and graphical interaction with simulations and their results. The first re-engineering project started in 2006 with the MIKE 11 engine. Later on, the MOUSE engine was merged into the MIKE 11 re-engineering project. The existing source code for MIKE 11 and MOUSE totals approximately 550,000 lines.

Meanwhile, the organisation was changing. DHI set up a software product department in order to strengthen the software development process and its design. As a consequence, the department decided to re-engineer the core computational parts of some of the one-dimensional simulation software products – MIKE 11, MOUSE and MIKE URBAN – into a project called MIKE 1D. The project is estimated to require 360 person-weeks merely for implementation.

Lately, the software product department officially promoted another project called the Decision Support System (DSS) Platform. The DSS Platform affords end users the leverage to customise ongoing water simulation using historical, current, and predictive data. The DSS Platform usually uses data that has already been gathered into persistent storage and occasionally works from operational data. The simulation it builds on has to be set up as well by developing the model of the water system.

The research cooperation with DHI addressed the introduction of product line architecture into product development. The basis for the research described here is the fieldwork which I have been involved in for two and a half years. I wrote a research diary documenting daily observations, interviews, and meetings. As a field worker, I was expected not only to observe, but also to influence the projects in which I participated. The research was designed as an action research by following CMD approach. The research activities are summarised in Table 1.

Table 1. Summary of research activities at DHI

Phase / Cycle	1.) MIKE 11 re-engineering project (August – November 2006)	2.) Merging of MIKE 11 and MOUSE engines re-engineering project (December 2006 – October 2007)	3.) MIKE 1D project (February 2007 – March 2009)
Participant observation	- Study functionalities and code architecture of MIKE 11 and MOUSE engines. - Compare between MIKE 11 and MOUSE engine source code. - Interview DHI staff members. - Found a striking similarity in the source code between MIKE 11 and MOUSE engines.	- Review of architectural documentation and online user references systems used at DHI. - Observe development practices and technical infrastructure of MIKE 11 and MOUSE engines. - Review off-the-shelf documentation generators. - Interview developers and internal users of MIKE 11 and MOUSE engines on how they can use the architecture document.	- Review off-the-shelf static code analysis tools. - Analyse MIKE 1D source code using the reviewed tools and identify the relative complexity of its components. - Compare the analysis with the previous cycle projects. - Join MIKE 1D project weekly meetings. - Interview MIKE 1D team members on the idea of assessing the architecture and how they can use of the architecture as an aspect of software development.
Deliberating change	- Present a poster highlighting identical code parts between MIKE 11 and MOUSE engines. - Present a talk on software architecture and product line architecture. - Participate in a sub-project on developing data access module architecture for the MIKE 11 re-engineering project.	- Propose a layered architecture to represent architectural knowledge. - Compare documentation generators and recommend a suitable one. - Update architecture documentation. - Create a prototype of an online architectural knowledge system.	- Conduct a workshop on architecture discovery with MIKE 1D team members. - Introduce the basic idea of architectural conformity checking. - Recommend suitable static code analysis tools. - Present the “good” and “bad” parts of the source code from the static code analysis tools. - Present an empirical study on architecture evaluation in industrial practice, the concept of software evolvability, and evolvability framework. - Propose Architecture-Level Evolvability Assessment (ALEA), see Chapter 10. - Organise a workshop on MIKE 1D and DSS Platform compatibility.
Evaluation	- Evaluate the flexibility of the data access module by looking at different change scenarios at DHI and their implications in terms of implementation efforts. - Found that organisation of software development influenced product line architecture development. - Identified 6 contextual dimentions to be taken into accout when evolving the architecture.	- Found that architectural knowledge was more visible in the discussion than in the document. - Found that the prototype of the online architectural knowledge system has been set up and used by developers and consultants at DHI headquater office.	- Found that architectural analysis tools and techniques embedded in daily routine were welcome by the development team. - Found that the development team uses “build hierarchy”, see Chapter 8, to check the compliance of their source code against the architecture’s structure when they build the software. - Validate ALEA and evolvability framework with MIKE 1D team members.

Due to a lengthy period of cooperation, research activities are chronologically divided into three cycles: (1) MIKE 11 re-engineering project, (2) merging of MIKE 11 and MOUSE re-engineering project, and (3) MIKE 1D project. Note that the research activities in the second cycle were finalised when the third cycle was under way. Each cycle consists of three phases (a, b, c on page 26). Most empirical evidence presented in this thesis is obtained from the last cycle. The DHI case will be presented again in Chapter 6, 8, 10 and 11. The DHI case reports the importance of architecture as a key for evolvability. The DHI case presents initial contextual dimension for evolvability that is later used for evaluating an evolvable architecture. The DHI case also reports how architecture is concretised in the development environment as a build hierarchy.

2.2. EASI-WAL case description

EASI-WAL is a government agency that was founded to simplify the communication between public entities and citizens or enterprises in Belgium’s Walloon region. EASI-WAL offers several software products and projects that support e-government and public administration at a local or regional level, e.g. a city council, regional government or parliament. This government agency was faced with the challenge of identifying generic parts, which are shared in a larger community, and specific/tailored parts, which are used by a single public body. Even if the specific/tailored parts will not benefit the larger community, the specific/tailored parts should be co-evolved with the generic parts at a reasonable cost and maintained by small teams [27, 33].

The first re-engineering project started with the College application that is currently called PloneMeeting—the official meeting management system for local or regional authorities. The research co-operation with EASI-WAL addressed the development of an automatic variability configurator for PloneMeeting and its related products. The configurator will be implemented by a group of researchers at the PReCISE research centre, University of Namur, Belgium [3]. The researchers apply variability modelling from feature diagrams and relate it to PloneMeeting source codes. Based on the diagram and the constraints of the PloneMeeting, they can then select various features. The variability realisation mechanism will be chosen and bound to the source code. Afterwards, the feature diagram reasoning will show the acceptance results.

The basis for the research is the fieldwork involving institutionalisation of variability management to the PloneMeeting project. While I was observing the project, I intervened by following the CMD idea even though I did not apply the approach formally. Further detail on the research method for EASI-WAL case will be presented in Chapter 7. However, the EASI-WAL case confirms the contextual dimensions, aka an evolvability framework that was a result of the early cycles of the DHI case. The EASI-WAL case also reports the concepts of software evolvability and socially embedded systems. Moreover, the EASI-WAL case emphasises the importance of the social perspective as well as the technical perspective in evolving a family of software products.

2.3. Interview study

This research was designed as an interview study, sampling eight software product development companies located in five different countries: Belgium, China, Denmark, Germany and Switzerland. Each company has its own ongoing software product developments. Sizes of the sampled companies range from three to more than twenty-thousand total employees. Interviewees were a mixed group and included a managing director, a chief technical officer, a chief architect, a marketing consultant, a group leader, and a number of software developers. Most of the interviewees did not want to disclose their personal or company names.

An interview guideline was prepared for facilitating the semi-formal interviews. The interview guideline had two parts: a series of free response questions, and a series of multiple-choice questions. The free response questions contained six categories: company introduction and interviewee, software architecture, co-operation, awareness, product line, and evolvability. The interviews—audio-taped and transcribed—were conducted from late 2007 till early 2008 with a duration that varied between thirty minutes and three hours. The transcription and analysis of the interviews were checked by the interviewees. Apart from that, we also provided confidentiality agreements for the interviewed companies.

Table 2. Workshops and interviews for evaluation

No.	Title	Month/Year	Participant
1	Prototype of data access module for MIKE11	Mar. 07	DHI
2	Architecture discovery workshop	Nov. 07	DHI
3	Tools for visualising dependency in architecture	Feb.- Jun. 08	DHI
4	Product line architecture of PloneMeeting	Mar. 08	EASI-WAL, PReCISE
5	Variability configurators	May 08	EASI-WAL, PReCISE
6	Introducing architecture awareness, product line architecture, architecture evaluation methods	Jul. 08	DHI
7	Architecture-Level Evolvability Assessment (ALEA) workshop	Feb. 09	DHI

The interview study is detailed in Chapter 9. The study highlights the aspects that we consider relevant for developing support for architectural practices for software product development. The important of the "walking architecture", "good reasons for bad documentation" indicate the need to develop social protocol fitting with local practices when introducing architecture representations and documentation, and we finally propose a means to promote architecture awareness.

3. Evaluation and credibility

Evaluation of prototypes, implementation efforts, tools and practices are part of the cooperative method development (CMD) which was mentioned earlier. Therefore, this section places emphasis on the credibility for the research.

In order to explicitly get confirmation on understanding real world practices, and get valuable comments and feedback from practitioners, many evaluation workshops were organised, as shown in Table 2.

The credibility or the trustworthiness of this research is based on how case studies and qualitative research interviews were performed and used. Examples for applying research methods were mentioned above, and with each chapter in Part II. However, the strategies to minimise possible threats to validity [50] shows as follows:

Prolonged involvement strategy permits the development of a trusting relationship between the researcher and the practitioners. The prolonged involvement strategy is used in the main case study. The study at the DHI Water Environment Health (DHI) was begun in August 2006 and finalised in March 2009. I will show my understanding of the work practices and that the evaluation is sound.

Triangulation. The basic idea of triangulation, as summarised in [54], is to gather different types of evidence to support a proposition or a hypothesis. Based on [41], triangulation provides an explicit vehicle for tracking the principle issues or limitations presented by a single empirical study within the field, and provides a solid scientific basis for deriving “fact” from a number or interlinked (by research question) empirical studies. A supplementary case study at EASI-WAL focusing on the PloneMeeting project was conducted to triangulate the first research question (*RQ1*). The supplementary case study is a triangulation for a concept and a framework for evolvability proposed in the main case study as well as the software type studied in this thesis. At the same time, qualitative research interviews on architecture awareness were conducted to triangulate the second and third research question (*RQ2* and *RQ3*). The interview study was a triangulation for the issue of architecture awareness and how people actually worked with architecture in development. In the end, the overall results were connected with, and contributed from, each empirical study presented in this thesis.

Peer debriefing and support refers to the role of peer support groups in qualitative research as a mechanism for debriefing and guarding against bias, for keeping the researcher “honest” throughout the study [44, p. 99]. In this research, the participation with different research groups/centres (Software development group, IT University of Copenhagen, Denmark; PReCISE research centre, University of Namur, Belgium; LERO research Centre, University of Limerick, Ireland) offers me an opportunity to get and give feedback and ideas to and from other students and researchers. The supplementary case study was collaborated with the PReCISE research centre. The results of the main and supplementary case studies support each other. The cooperation with Dr. Wolf-Gideon Bleek from University of Hamburg, Germany supports setting up the interview study and introducing architectural tools and practices in the main case study.

Leaving an **audit trail** means adopting a spirit of openness and documenting each step taken in data collection and analysis [44, p. 101]. The audit trail strategy is not intended for exact replication, but it is a way to enhance another researcher to be able to use the audit trail to reproduce and verify the finding. In this research, a research diary documenting daily observations, informal interviews, and meetings for two case studies were written continuously. Formal interviews conducted in the case studies, the interview study, and the workshops shown in Table 2 were audiotaped and transcribed. The drawing artefacts and diagrams on the whiteboard were photographed.

Member checking is a very valuable means of guarding against researcher bias [50, p. 175]. In this research, many evaluation workshops with specialists at DHI and EASI-WAL were conducted. Articles were reviewed by practitioners, with respect to studies, before they were published. All interviewees of software architecture awareness were required to check the transcription of their interviews before the data was analysed.

4

RESEARCH RESULTS

The research discussed in this thesis aims at introducing an engineering discipline to support evolvability of software products. The introduced engineering discipline is empirically grounded; it is not just another academic method, but is applied and evaluated with respect to industry. This chapter shows research outcomes. Section 1 presents a collection of articles that answer the research questions discussed in the introduction of this thesis. Section 2 presents related articles which were produced during the research. Section 3 discusses the overall results for this thesis. Section 4 notes a by-product of this thesis.

1. List of articles

This section presents six articles in which the outcomes and contribution to this thesis are explained together with the answers to the research questions. The research questions (*RQs*) shown in Chapter 1 are as follows:

RQ1: What is software evolvability?
RQ2: Do software developing companies manage to maintain evolvability of their software products over a long lifetime, and how do they do that?
RQ3: How to develop software products for evolvability?

Article 1.

H. Unphon and Y. Dittrich, "Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture." Innsbruck, Austria: IASTED International Conference on Software Engineering, 2008, pp. 178–183.

The first article presented in Chapter 6 of this thesis shows empirical evidence of organisation and business domains alongside technical aspects influencing the development of product line architecture. The empirical evidence was categorised into six contextual dimensions around architecture: business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection. The *business context* is the context or environment to which the system belongs. The *use context* relates the system to the work practices of the intended users. The *software engineering organisation* is the organisational context in which the software development is carried out. The *software engineering practice* refers to the analysis of the work practices of the system developers. The *technical infrastructure* lists the hardware and basic software assets backing the system. The *technical selection* is part of a suggested design and needs to be seen in the context of existing and planned systems, as well as in the context of other systems that are part of the same design. The empirical evidence presented in the main case study results in answering *RQ1* and addresses *RQ3*, and partly answers *RQ2*.

Article 2.

H. Unphon, Y. Dittrich, and A. Hubaux, "Taking Care of Cooperation when Evolving Socially Embedded Systems: The PloneMeeting Case." Vancouver, Canada: The Cooperative and Human Aspects of Software Engineering 2009 (CHASE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), May 2009.

The second article presented in Chapter 7 of this thesis is empirical evidence from the supplementary case study which supports the findings in the main case study. This article proposes an evolvability concept and an evolvability framework. The evolvability concept defined in this article is *the adaptability of software in order to serve the needs of use and business contexts over time reflecting on its architecture*. The evolvability framework is a successive of six contextual dimensions developed in the first article. The usage of the evolvability framework in this article is to analyse and support the understanding of change during their evolutions. This article contributes to answering *RQ1* and directs to *RQ2*.

Article 3.

H. Unphon, "Making Use of Architecture throughout the Software Life Cycle—How the Build Hierarchy can Facilitate Product Line Development." Vancouver, Canada: The Forth Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), May 2009.

The third article presented in Chapter 8 of this thesis shows an empirical study of how architecture is actually used beyond the design phase of product line development. The article presents a *build hierarchy*[1] that complements the agile development practice in the main case study. This article contributes to answering *RQ2* and *RQ3*.

Article 4.

H. Unphon and Y. Dittrich, "Architecture awareness," 2009, submitted to the journal of Systems and Software.

The sixth article presented in Chapter 9 of this thesis is the interview study with eight product developing companies on the daily use of architecture. Results of the study indicate that a chief architect or central developer acts as a 'waling architecture' devising changes and discussing local designs while at the same time updating his own knowledge about problematic aspects that need to be addressed. Architecture documentation and representations might not be used, especially if they replace the feedback from ongoing developments into the 'architecturing' practices. Referring to results from Computer Supported Cooperative Work (CSCW), we discuss how explicating the existing structure needs to be complemented by social protocols to support the communication and knowledge sharing processes of the 'walking architecture'. This article broadens and refines the answer for *RQ2* and *RQ3*.

Article 5.

H. Unphon, "Architecture-Level Evolvability Assessment," 2009, submitted to Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009.

The fourth article presented in Chapter 10 of this thesis proposes a method for Architecture-Level Evolvability Assessment (ALEA). The evolvability framework proposed in the second article is applied to the ALEA method in order to propagate the effect of architectural changes. The ALEA method has been validated in the main case study. The evolvability concept was refined by taking *sustainability* along with

[1] Build hierarchy is a technique to represent components and organise a series of generating executable code based on dependencies between the components.

Table 1. A summary of articles contributing to research questions

	RQ1	*RQ2*	*RQ3*
Article 1.	- Evolvability is not only about technical adaptability but it is also about the importance of future design supporting the work of organisation, business context, and work practice on organisation and developmental context. - The first version of evolvability framework.	- Maintenance and re-engineering the structure of software product. - Developing product line architecture.	- Architecture must be designed for evolvability.
Article 2.	- The first version of evolvability concept. - The second version of evolvability framework.	- Refactoring the structure of software products. - Developing variability management for the product family.	
Article 3.		- Architecture is used beyond design phase.	Thinking in terms of architecture is embedded in an integrated development environment as a build hierarchy.
Article 4.		- Walking architecture is a person or a group of people that is responsible for architecture related activities.	- An engineering discipline for evolvability must promote architecture awareness as collaboration mechanism in development practice.
Article 5.	- The second version of evolvability concept.		- Evolvability framework is used in Architecture-Level Evolvability Assessment (ALEA) method in order to systematically analyse architecture.
Article 6.	- A summary of evolvability concept and framework.		- The engineering discipline for evolvability brings together conceptual, human and technical dimensions of software development.

adaptability of architectural changes. This article contributes to answering *RQ3* and refines the answer for *RQ1*.

Article 6.

H. Unphon, "Introducing an evolvable product line architecture," 2009, submitted to the journal of Empirical Software Engineering.

The fifth article presented in Chapter 11 of this thesis proposes an engineering discipline for maintaining software evolvability that is suitable for industrial practice. This article summarises overall results from the main case study of the research. The proposed engineering discipline completes the answer for *RQ3* and summarises the answer for *RQ1*.

Table 1 summarises how each of above articles contribute to answering the research questions.

2. Other articles

The following articles relate to the research discussed in this thesis, but not included in this thesis.

H. Unphon. (2007) Comparison of documentation generators for C#. Technical note. [Online]. Available: http://www.itu.dk/people/unphon/technical_notes/CDG_v2007-05-15.pdf.

This article presents and reviews general and technical information for a number of documentation generators which are mainly used for C# and .NET Framework 2.0and makes suggestions according to the given criteria from the main case study.

A. Hubaux, P. Heymans, and H. Unphon, "Separating variability concerns in a product line re-engineering project," in Proceedings of the Early Aspects Workshop at AOSD'08, Brussels, Belgium, 2008.

This article gives an opportunity to collaborate with the supplementary case study, and presents the process for eliciting the variability of PloneMeeting, an Open Source project, and reports the initial results obtained when applying variability modelling techniques promoting separation of concerns between software variability and product line variability.

H. Unphon. (2008) A comparison of variability modelling and configuration tools for product line architecture. Technical note. [Online].Available: http://www.itu.dk/people/unphon/technical_notes/CVC_v2008-06-30.pdf.

This article presents a number of variability modelling and configuration tools for product line architecture that are currently available. The article categorises the comparisons into general information, technical infrastructure, operating systems support, rendering of modelling, format of input/output models support, modelling and configuration functionalities, and development functionalities. These categories and their corresponding criteria are based on the uses for PloneMeeting, an Open Source software family on eGovernment web applications. The purposes of this article are a guideline toward institutionalizing variability modelling and configuration tools to the PloneMeeting product family. There is supplementary material of the lecture on feature modelling given by the PReCISE research centre [3].

H. Unphon, M. A. Babar, and Y. Dittrich, "Identifying and Understanding Software Architecture Evaluation Practices," Technical report (work in progress), 2009.

The goal of this article is to describe design, logistics, and findings of an empirical study aimed at identifying and understanding different aspects of software architecture evaluation practices in industry. The results of this study are expected to provide useful insights into software architecture evaluation practices based on the experiences and perception of architects who regularly evaluate software architecture in various size applications.

3. Summary of research results

The research contribution is an engineering discipline for enhancing software evolvability. The research based on the main case study presents the success of introducing software product lines as a new software development approach for re-engineering legacy software products. The research supported by the supplementary case study confirms the understanding of evolvability. The engineering discipline proposed in the research combines conceptual, human, and technical dimensions in such a way that the new software development approach does not jeopardise the continuous evolvability of the re-engineered software products. The research emphasis is on the role of architecture for maintaining evolvability. Empirical study presented in this thesis shows that oftentimes what the literature or the software engineering textbooks recommend underestimates the industrial practice, e.g. claiming that an architecture for a software system does not really exist except in its documentation [8, 35]. The study reveals one of many reasons that future research can make use of software architecture's industrial practices in a better way.

The research results have iteratively refined the answers to the research questions as follows:

RQ1: What is software evolvability?

Based on the empirical research, software evolvability is defined as technical adaptability and sustainability with respect to the use and business contexts as well as the development organisation and development practice reflecting on its architecture. Article 1 identifies six contextual dimensions (i.e., business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection) that influence its architecture's adaptability and sustainability. The *business context* is the context or environment which the system is used. The *use context* relates the system to the work practices of the intended users. The *software engineering organisation* is the organisational context in which the software development is carried out. The *software engineering practice* refers to the analysis of the work practices the system develops. The *technical infrastructure* lists the hardware and basic software assets backing the system. The *technical selection* is part of a suggested design and needs to be seen in the context of existing and planned systems, as well as in the context of other systems that are part of the same design.

The evolvability framework presented in Figure 1 visualises interrelationships among the contextual dimensions; applying changes in one dimension induces changes in the other dimensions. The supplementary case study presented in this thesis confirms that the framework can be applied in order to understand changes and their effects for the socio-technical context [62].

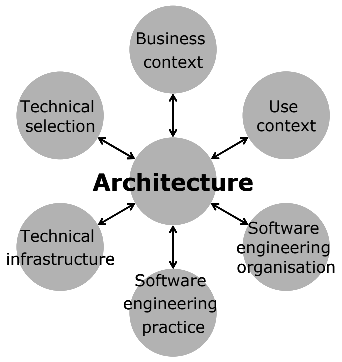


Figure 1. Evolvability framework

RQ2: Do software developing companies manage to maintain evolvability of their software products over a long lifetime, and how do they do that?

Yes, they do. When the needs of use and business contexts trigger changes to the software, many decisions are taken at architecture-level. The research from the main case study looks into the developmental practice at DHI. It indicates that the observed practices deviate from what is proposed in many textbooks. To better understand this practice and to see how widespread it is, an interview study on architecture awareness was designed. The interview study resulted in Article 4.

The research presented in this thesis proposes architecture awareness as a way to develop support for architectural practices for software product development. The structure of the software product is regarded as an important asset of the development. But rather than documenting it in a formal way, most companies rely on a '*walking architecture*', a key person or a number of key persons who maintain and update the structure of the software, who are involved in the discussions of changes motivated in the development, or by new requirements, and who introduce new developers to the structure of the software. Representations of the architecture, thus are temporary and partial: sketches on whiteboard and scrap paper used in a specific situation. The result of this practice is not only the distribution of architectural knowledge to the development team, but also an update of the chief architect's knowledge on the issues the developers discuss during implementation. This is more than what practitioners can get from traditional architecture documentation, which might be a '*good reason*' for what academia may call '*bad documentation*' practice.

Introducing a more explicit architecture practice at DHI led to the research documented in Article 3. A build hierarchy representing the static architecture is implemented in developers' integrated development environment in order to mediate the architecture conformance checking between design architecture and code architecture. The divergence between the design architecture and the code architecture will be reported within ten minutes after introducing and resolved within a day. Besides, an open workspace also promotes information flow (e.g., updating architecture knowledge) between team members within the same physical location.

RQ3: How to develop software products for evolvability?

The research results point out the importance of considering both the social and organisational as well as the technical dimensions so that the continuous evolution should not be jeopardised. Furthermore, the success of evolvable software product development is based on a synergy between conceptual, human, and technical levels: the *conceptual level* provides concepts for architectural design, architectural processes and evaluation that promotes the coherent perception of the state of the practice; the *human level* refers to work practices and social interaction of teams and stakeholders; and the *technical level* refers to a system or technical infrastructure.

To illustrate this point, we use results from our empirical study (two case studies and an interview study). At the conceptual level, an evolvability framework is given as an example. The framework helps practitioners and us, as researchers, to understand how design happens. An example at the human level is the Architecture-Level Evolvability Assessment (ALEA) method. ALEA method helps project members to structure their discussions in such a way that the members take into account adaptability as well as sustainability when evaluating architectural changes. At the technical level, a build hierarchy is an example of an architecturing technique for agile development. However, success on the technical level can only be achieved through support and interaction at the human level. This is confirmed by the interview study. The interviewees' reports from their practice indicated the need to complement tools and representations with social protocols so that they become useful for architectural practice.

4. The by-product of the thesis

While the main case study was progressing, the interview study with product developing companies was started. To begin with, the ideas I had initially did not seem to fit what I was hearing or observing from practitioners. My ideas of software architecture were radically changed in relation to practitioner's experiences. Based on this study, the term architecture awareness has been proposed. Architecture awareness focuses on the daily use of architectural knowledge: (*i*) how to make changes to a module that might have implications on other's code – that is that change the interface – visible, (*ii*) how to monitor changes that are relevant for the task at hand, and (*iii*) how to monitor changes to the code, the requirements and the context that makes it necessary to change the architecture and thus change the design and implementation of the different modules.

Initially, I discussed the design of the system with project members of the main case study. They showed me source code and function calls. At that time, the project members thought that the concept of architecture was too abstract, a piece of outdated drawing on a wall, or something that they could not run on their computer's processors. After two and a half years of the co-operative research project, the thinking in terms of architecture and product line architecture becomes a trustworthy solution for practical and real-scale problems.

The gap between academic research and industrial practice needs continuous attention. Looking at software engineering from a work practice perspective, it is interesting to learn how non-software engineers, or people who were educated in some other field (e.g., mathematics, physics, or hydraulics), do software development. I sometimes found that there was no software engineering discipline in an industrial context. Then, an interesting point arose about how the non-software engineers managed their work practice regardless.

To summarise, the research discussed in this thesis takes an academic approach and makes it relevant to practitioners, for instance, in the case studies. At the same time, the research takes evidence from the practitioners (e.g. the interview study) and makes sense of the evidence in order to establish research outcome grounded in real-life scenarios. The research presented in this thesis goes beyond the related research (Chapter 2) in the following way. The research proposes an evolvability framework grounded upon empirical evidence. Based on the evolvability framework, the research introduces ALEA method that extends adaptability with the complement of sustainability in order to evaluate the architecture. In addition, the research focuses on architecture as a product and architecturing as a process. Using the notion awareness mechanism as a social protocol derived from Computer Supported Cooperative Work (CSCW), this thesis implements architectural tools and practices that are integrated with the development practices in order to promote architecture awareness.

5

CONCLUSIONS

This chapter summarises the research discussed in this thesis and reports limitations and future research, as shown in Section 1 and 2, respectively.

1. Thesis summary

The goal of this research is to propose engineering discipline for enhancing continuous evolvability. The research is designed as a qualitative empirical study using two case studies (i.e., DHI Water Environment and Health (DHI) [2] and EASI-WAL [1]) that re-engineer their software products using software product line approach [21], and an interview study with different product developing companies.

DHI is an independent research and consultancy organisation providing various commercial simulation software products for water and environment. The research described in this thesis firstly involved the re-engineering project of MIKE 11—the river modelling system dealing with surface water problems. For a supplementary case, EASI-WAL is a government agency providing several open-source software products for public bodies in Belgium's Walloon region and with Belgium's French speaking communities. The research involved a recently released version of PloneMeeting—the official meeting management system for local and regional authorities, which applies the product line engineering approach to an open-source software product family.

The two case studies are addressed as re-engineering legacy software products into software product lines approach. The main difference between these cases and prior cases of software product lines is the type of software products, which the research described in this thesis calls "*socially embedded systems*" [62]. Due to the dynamics of software evolution [40], dynamics of usage cannot be captured as variability requirements. Organisational and work practice aspects of the use context as well as the organisation and work practice of the development context have to be considered for the architectural design. This entails not only understanding design decisions, but also improving the day-to-day software development process that enhances evolvability by re-engineering existing software products.

The interview study reveals that, in everyday development practices, software architecture is a medium of discussion and evaluation for design decisions even on a small scale. For instance, the experienced software engineers explain their understanding to novices by informal sketches and references to source code. On the continuous evolution, the effects of change that are not in line with the original design rational may diffuse slowly and steadily in the architecture of software products. The research discussed in this thesis proposes ways to promote architecture awareness as an engineering discipline.

2. Limitation and future work

The research discussed in this thesis is rather biased toward the main case study at DHI. Although, the supplementary case study at EASI-WAL was conducted for three months, it did not triangulate all the concepts and engineering disciplines that were later proposed in the main case study. Thus, the concepts and disciplines should be introduced to and studied in other product developing companies as well. Moreover, validating and refining the proposed concepts and framework should be addressed in future research. In my view, software evolution is truly a software engineering epic that continues with unanswered questions, and will need more contribution from future researchers to complete and correct them.

The future researcher should continue to address how practitioners manage their practice and how to support those practices as opposed to "what would make sense for us as researchers?" In the beginning of my fieldwork, I as a researcher tried to "educate" practitioners. They quickly rejected my attempts, for they felt my advice was difficult to understand, or would waste their time learning an approach that did not fit their work practice. Since then, I always try to understand their methodologies before I "introduce" mine to them.

References

[1] Commissariat EASI-WAL. [Online]. Available: http://-easi.wallonie.be/xml/

[2] DHI Water Environment Health. [Online]. Available: http://-www.dhigroup.com

[3] The PReCISE research centre. [Online]. Available: http://-www.fundp.ac.be/universite/interfacultaire/precise/

[4] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 244–254.

[5] M. Ali Babar, T. Ihme, and M. Pikkarainen, "An industrial case of exploiting product line architectures in agile software development," in *accepted in the 13th International Conference on Software Product Lines*, San Francisco, USA, 2009.

[6] M. A. Babar and I. Gorton, "Software Architecture Review: The State of Practice," *Computer*, vol. 42, no. 7, pp. 26–32, 2009.

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[8] L. Bass and R. Kazman, "Architecture-based development," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-99-TR-007, 1999. [Online]. Available: www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr007.pdf

[9] J. Bayer, J. Girard, M. Wuerthner, J. M. DeBaud, and M. Apel, "Transitioning legacy assets to a product line architecture," in *Proceeding of the Seventh European software Engineering Conference (ESEC'99), Lecture Notes in Computer Science 1687*, Toulouse, France, September 1999, pp. 446–463.

[10] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[11] L. Belady and M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 1, pp. 225–252, 1976.

[12] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (ALMA)," *J. Syst. Softw.*, vol. 69, no. 1-2, pp. 129–147, 2004.

[13] P. D. Borches and G. M. Bonnema, “On the origin of evolvable systems: Evolvability or extinction,” in *Proceedings of the TMCE 2008*, I. Horváth and Z. Rusák, Eds., Kusadasi, Turkey, April 21–25 2008.

[14] J. Bosch, *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[15] H. Breivold, I. Crnkovic, and P. Eriksson, “Analyzing software evolvability,” in *Computer Software and Applications, 2008. COMPSAC ’08. 32nd Annual IEEE International*, Turku, Finland, 2008.

[16] F. Brooks and K. Iverson, *Automatic Data Processing (System 360 Edition)*. John Wiley, 1969.

[17] L. Brownsword and P. Clements, “A case study in successful product line development,” Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-96-TR-016, 1996.

[18] R. Carbon, M. Lindvall, D. Muthig, and P. Costa, “Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design,” in *proceedings of the 1st International Workshop on Agile Product Line Engineering (APLE’06)*, Kyoto, Japan, 2006.

[19] H. B. Christensen, K. M. Hansen, and K. R. Schougaard, “Ready! set! go! an action research agenda for software architecture research,” in *WICSA ’08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 257–260.

[20] J. A. Christian III, “A Quantitative Approach to Assessing System Evolvability,” NASA Johnson Space Center, NASA Johnson Space Center, Houston, TX 77058, Tech. Rep., 2004.

[21] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[22] P. Clements, “Active Reviews for Intermediate Designs,” SEI, Carnegie Mellon University, Tech. Rep. CMU/SEI-2000-TN-009, 2000.

[23] A. Cockburn, *Agile Software Development*, ser. Agile Software Development Series. Addison-Wesley Professional, October 2001.

[24] S. Cook, H. Ji, and R. Harrison, "Software Evolution and Software Evolvability," Working paper, 2000, university of Reading, UK.

[25] J. Corbin and A. Strauss, *Basic of Qualitative Research; Technique and Procedures for Developing Grounded Theory*, 3rd ed. USA: Sage Publications, 2008.

[26] J. Dager, *Software Product Line: Experience and Practice*. Kluwer Academic Publisher, 2000, ch. Cummin's Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls, pp. 23–45.

[27] G. Delannay, K. Mens, P. Heymans, P.-Y. Schobbens, and J.-M. Zeippen, "PloneGov as an Open Source Product Line," in *Proceedings on the Third International Workshops on Open Source Software and Product Lines*, Kyoto, Japan, September 2007.

[28] Y. Dittrich, K. Rönkkö, J. Eriksson, C. Hansson, and O. Lindeberg, "Cooperative method development," *Empirical Software Engineering*, vol. 13, no. 3, pp. 231–260, 2008.

[29] C. Floyd, R. Keil-Slawik, R. Budde, and H. Zullighoven, Eds., *Software Development and Reality Construction*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1992, illustrator-Weiler-Kuhn, C.

[30] D. Garlan and D. Perry, "Introduction to the special issue on software architecture," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 269–274, 1995.

[31] B. G. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago: Aldine, 1967.

[32] G. K. Hanssen and T. E. Fægri, "Process fusion: An industrial case study on agile software product line engineering," *J. Syst. Softw.*, vol. 81, no. 6, pp. 843–854, 2008.

[33] A. Hubaux, P. Heymans, and H. Unphon, "Separating variability concerns in a product line re-engineering project," in *Proceedings of the Early Aspects Workshop at AOSD'08*, Brussels, Belgium, 2008.

[34] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2000-TR-004, ADA382629, 2000. [Online]. Available: http://www.sei.cmu.edu/-publications/documents/00.reports/00tr004.html

[35] R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, and S. G. Woods, "Experience with performing architecture tradeoff analysis," in

ICSE ’99: Proceedings of the 21st international conference on Software engineering. New York, NY, USA: ACM, 1999, pp. 54–63.

[36] R. Kazman, L. Bass, G. Abowd, and M. Webb, “SAAM: A Method for Analyzing the Properties of Software Architectures,” in *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 81–90.

[37] P. Kruchten, *The Rational Unified Process: An Introduction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[38] M. M. Lehman, “Laws of software evolution revisited,” in *EWSPT ’96: Proceedings of the 5th European Workshop on Software Process Technology*. London, UK: Springer-Verlag, 1996, pp. 108–124. [Online]. Available: http://www.doc.ic.ac.uk/~mml/feast2/papers/-pdf/556.pdf

[39] M. Lehman, “On Understanding Law, Evolution, and Conservation in the Large-Program Life Cycle,” *Systems and Software*, vol. 1, no. 3, pp. 213–231, 1980.

[40] M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.

[41] J. Miller, “Triangulation as a basis for knowledge discovery in software engineering,” *Empirical Softw. Engg.*, vol. 13, no. 2, pp. 223–228, 2008.

[42] H. Muccini and A. V. D. Hoek, “Towards Testing Product Line Architectures,” in *International Workshop on Testing and Analysis of Component Based Systems*, 2003, pp. 111–121.

[43] P. Naur, “Programming as Theory Building,” *Microprocessing and Microprogramming*, vol. 15, pp. 253–261, 1985.

[44] D. Padgett, *Qualitative methods in social work research: challenges and rewards*. SAGE Publications, 1998.

[45] D. Parnas, “Information distribution aspects of design methodology,” in *Proceedings of the 1971 IFIP Congress*, North Holland, 1971.

[46] D. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[47] D. Parnas, “On a ’Buzzword’: Hierarchical Structure,” in *Proceedings of the 1974 IFIP Congress*. Kluwer, 1974.

[48] D. Parnas, “On the design and development of program families,” *IEEE Transactions on Software Engineering*, vol. 2, no. 1, 1976.

[49] G. S. Percivall, “System Architecture for Evolutionary System Development,” in *Proceedings of the 4th Annual Symposium of the National Council on Systems Engineering*, 1994, pp. 571–575.

[50] C. Robson, *Real world research: a resource for social scientists and practitioner-researchers*, 2nd ed. UK: Blackwell publishing, 2002.

[51] D. Rowe, J. Leaney, and D. Lowe, “Defining Systems Evolvability - A Taxonomy of Change,” *Engineering of Computer-Based Systems, IEEE International Conference on the*, vol. 0, p. 0045, 1998.

[52] W. W. Royce, “Managing the development of large software systems: concepts and techniques,” in *ICSE ’87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 328–338.

[53] K. R. Schougaard, K. M. Hansen, and H. B. Christensen, “SA@Work,” *Asia-Pacific Software Engineering Conference*, vol. 0, pp. 411–418, 2008.

[54] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[55] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[56] IEEE Computer Society, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE-SA Standards Board Std., September 2000.

[57] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science Series)*, 8th ed. Addison Wesley, June 2006.

[58] D. Soni, R. L. Nord, and L. Hsu, “An empirical approach to software architectures,” in *IWSSD ’93: Proceedings of the 7th international workshop on Software specification and design*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 47–51.

[59] K. Tian and K. Cooper, “Agile and software product line methods: Are they so different,” in *proceedings of the 1st International Workshop on Agile Product Line Engineering (APLE’06)*, Baltimore, Maryland, USA, 2006.

[60] H. Unphon, “Making Use of Architecture throughout the Software Life Cycle—How the Build Hierarchy can Facilitate Product Line Development.” Vancouver, Canada: The Forth Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), May 2009.

[61] H. Unphon and Y. Dittrich, “Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture.” Innsbruck, Austria: IASTED International Conference on Software Engineering, 2008, pp. 178–183.

[62] H. Unphon, Y. Dittrich, and A. Hubaux, “Taking Care of Cooperation when Evolving Socially Embedded Systems: The PloneMeeting Case.” Vancouver, Canada: The Cooperative and Human Aspects of Software Engineering 2009 (CHASE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), May 2009.

[63] H. Unphon, “Architecture-Level Evolvability Assessment,” 2009, unpublished results.

[64] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag Berlin Heidelberg, 2007.

[65] H. Yang and M. Ward, *Successful Evolution of Software Systems*. Norwood, MA, USA: Artech House, Inc., 2003.

PART II

ORGANISATION MATTERS: HOW THE ORGANISATION OF SOFTWARE DEVELOPMENT INFLUENCES THE DEVELOPMENT OF PRODUCT LINE ARCHITECTURE

Hataichanok Unphon, Yvonne Dittrich

IT University of Copenhagen
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{unphon, ydi}@itu.dk

ABSTRACT. Our work aims at understanding the design rationale for product line architecture by focusing on the design of common data access modules for complex simulation software products. This paper presents empirical evidence of organisational and business domain aspects that influence the development of product line architecture. We suggest that the assessment of use-situation and history of organisational structure should be considered when creating product line architectures, especially for products that are tailored and used interactively.

KEY WORDS

Product line architecture, software development organisation

1. Introduction

Software architecture and product line architecture (PLA) are normally treated as purely technical issues by means of making the encompassing architecture for the products line. Many industrial experiences in PLA ([1], [2], [3], [4]) investigated into technical embedded systems. The architecture specifies 'what is common and what are variations' that are explicitly allowed among them. When starting with a research project around the design of simulation software we entered the cooperation with similar expectations. However, by taking part in the design of a first step towards PLA, our experience has taught us that organisation matters when we deal with a software product. This article discusses different organisational aspects that influence the development of a product line for non-technical embedded systems. We conclude that when introducing a PLA not only technical but also organisational and business domain aspects have to be taken into account, especially for products that are tailored and used interactively.

Others also have discovered the relationship between organisation and software architecture: when a structure of the system is difficult to derive from low-level facts, e.g. source code, an organisational structure (organisation of the development groups that creates the system) might give insightful information according to Conway's law [5]. Furthermore, Ganesan et. al. [1] have shown that recovered ownership architecture, which captures the relationship between the developers and the software components they work on, matches very well with the implemented architecture of product line. Although their definition of ownership works well in their case study, they cannot generalize the definition because it is based only on the commit history—the account of event that occurred when one's changes to a working copy of source code which are reflected in the repository of Concurrent Versions System (CVS).

Several influences between organisational structure and PLA became visible when we tried to understand the architecture during a re-engineering project at the DHI Water Environment Health (DHI), an independent research and consultancy organisation providing various simulation software products for the water and environment. This paper illustrates how organisation and business domains have a bearing on producing PLA for an end-user customised system.

The next subsection discusses terms organisation and PLA to make clear the fundamental concepts of this paper. The rest of the paper is outlined as follows: Section 2 describes our case at DHI, the simulation software, and the research methodology; Section 3 illustrates how organisation and business domain matter in producing PLA by giving citation on our field material; Section 4 presents related works, i.e. methods for producing PLA, organising for software product lines, and successful case studies; Section 5 covers the conclusions and future work.

1.1. Organisation and product line architecture: an overview

To initiate a discussion on the relationship between organisation and product line architecture (PLA) it is first to define what is meant by each of the two terms.

From an organisational studies perspective, organisations are defined as social arrangements for the controlled performance of collective goals [6]. In a broad sense, therefore, examples of an organisation may include a group of C# programmers, a software development team, the Water Resource Department, or the DHI.

Now, what is meant by the term ‘product line architecture’? Product line is a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission area [7]. But what is ‘architecture’? Well, again there is no single answer to this question. A suitable working definition for the purpose of this paper is that, in essence, the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [8]. According to these definitions, product line architecture is a single specification capturing the overall architectures of a series of closely related products [9].

2. Case descriptions and methods

This section gives an overview of the company where we conducted our research, describes the simulation software, and explains our research methodology.

2.1. DHI Water Environment Health (DHI) study

The DHI Water Environment Health (DHI) is an independent research and consultancy organisation that was founded to promote technological development in areas relevant to water, environment and health in the field of ecology [10]. The organisation develops a wide range of software systems for complex simulations, based on well-established computational kernels for solving partial differential equations. After nearly 30 years of successful use and development, however, the maintenance of the software was increasingly time consuming task. The company was faced with the challenge of identifying and developing a kernel for data handling, simulation setup, and graphical interaction with simulations and their results. For this reason the idea of re-engineering the existing systems was initiated. The first re-engineering project started with the MIKE 11 system —the river modelling system dealing with surface water problems.

2.2. The simulation software

MIKE 11 was designed as a single-user desktop application for one-dimensional models of river systems. On an abstract level the process of the complex simulation conducts as follows: *i.*) a set of input data is read from a *file*, *ii.*) simulations based on that data are performed, and *iii.*) a set of output data is produced.
The architecture for handling data (see Fig. 1) has two parts: *Setup part*, handling setup data, and *Result part*, respectively handling result data. The *Setup part* has its own Graphical User Interface (*GUI*), which is called *Setup GUI. Setup GUI* is used by an end-user in order to enter

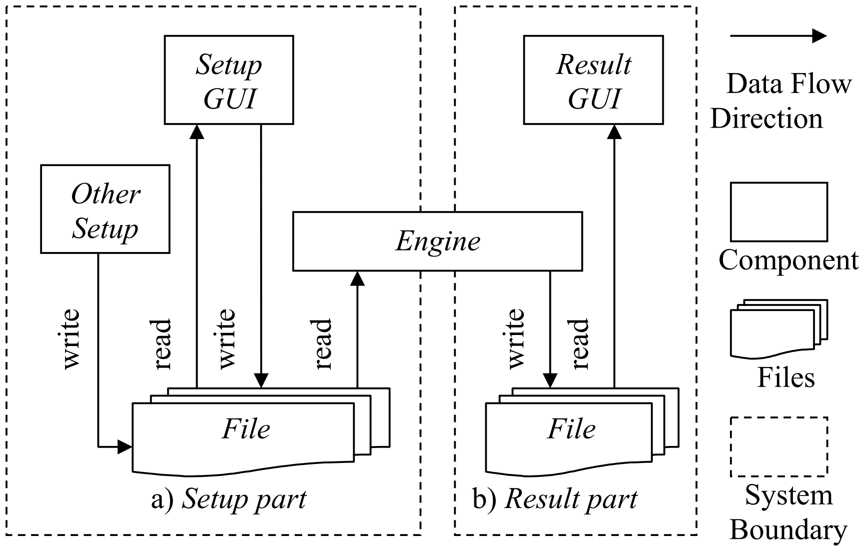


Figure 1. Run-time architecture of traditional data access

a set of input data that will be stored in a *file*, which can be ASCII or binary format depending on geographical data types. The *Setup GUI* is able to retrieve existing input data from the *file*. Other programs might write the setup data directly to the *file* or database (*Other Setup)*. An *Engine* is a basic part of the simulation. It reads the input data from the given *file* and performs the simulation. It produces a simulation result that will be stored in another *file* format. The *Result part* has its own *GUI*, which is called *Result GUI*. *Result GUI* is used by an end-user in order to read a set of output data which is stored in the *file*. The architecture of the *Setup part* and the *Result part* are shown in Fig. 1a and 1b respectively.

As illustrated in Fig. 1, each *GUI* and *Engine* pair handles the same *file*, but since they run separately they have independent pieces of code handling the data access. It is noticeable that most of these code pieces in the *Engine* and the *GUI* are closely related with the respective ones in the other modules. From the software developers' point of view, the main problem is trying to maintain three copies when changing the data format. In addition, complex simulation software products also encounter this problem even though the database is used as data storage instead of a *file*. Another problem is the lack of openness, or the inability to use hard coding, or programs, e.g. the third party (*Other Setup*). Undeniably, however, these were old problems in software development for a few decades ago.

Besides the MIKE 11 software, DHI also develops and maintains two other branches of software for simulation on one-dimensional systems: MOUSE and MIKE Urban, the later uses ArcGIS® technology [11], for modelling of urban sewers. Though MOUSE and MIKE Urban use databases for transferring geographical data between *GUI* modules and the simulation *Engine* in addition to *file*s, the same problems of maintaining duplicate and triplicate code are recognised.

2.3. Research methodology

The research cooperation with DHI addressed the redesign of the above described data access part of the MIKE 11 software. The basis for the research described here is the fieldwork by one of the authors involved in the MIKE 11 re-engineering project who wrote a research diary documenting daily observation, informal interviews, and meetings.

Additionally, the transcript of recordings of one of the workshops was also analysed. From these materials, we derived several categories of factors that influenced producing PLA. As the field worker was expected to not only observe, but to also influence the re-engineering project, the research was designed as action research by following the co-operative method development approach (CMD) [12].

- **Participant observation:** Initially, we began with participant observation of the MIKE 11 re-engineering project. We studied functionality of the existing application, structure of the source code, and its relevant product, i.e. MOUSE. We also compared the similarity of source code between MIKE 11 and MOUSE. In the mean time, we conducted informal interviews with the DHI staff members.

- **Deliberating change:** After we found a striking similarity in the source code between MIKE 11 and MOUSE, we presented a poster highlighting identical code parts in the corridor in order to initiate discussion among software developers. We subsequently had a presentation on software architecture and PLA, which the ideas of cooperation between departments became visible. Later, we participated in a subproject that mainly discussed a new data access module and developed a prototype of the module.

- **Evaluation of the prototype and implementation efforts:** Finally, we organised an evaluation workshop with a group of DHI software specialists. We evaluated the flexibility by comparing the new design of the data access module with the old design, which did not have the data access modules. We also looked at different change scenarios at DHI and their implication in terms of implementation efforts. Apart from that, we continued with participant observation of the data access module subproject, a joint meeting between departments, and the informal interviews with DHI staff members.

3. Organisation and business domain matters

When analysing the evaluation workshop and the discussions around the implementation of the new structure the influence of a series of organisational factors rather than technical design considerations became visible. We structure an analysis into two subsections. The first subsection is a reflection on the technical design, which focuses on rationale for creation of the old design. The second subsection explains influence on the re-engineering project, which discusses our research and analysis of the new design proposed by the project. This design attempts to minimise the maintenance effort when accessing data.

3.1. Reflection on the technical design

Based on the analysis of our field material we categorised the reflection on the technical design into a business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection aspects.

Business Context: DHI is a pioneering organisation that develops software application for hydraulic modelling. In 1972, System 11 and System 21 were the first computational models developed to simulate water flow patterns with the help of one-dimensional and two-dimensional models. A three-dimensional simulation was developed in the 1980s. Originally, the organisation was founded on hydraulic research characteristic, not on software engineering. Software development and software maintenance were challenges on only a small scale. In the late 1970s, DHI developed System 11 to MIKE 11, MOUSE following the consultancy projects and the market requests of open channel and pipe network. In the last decade, MIKE11 and MOUSE were developed in parallel because the ownership was spitted. MIKE Urban (MU) was initiated 3-4 years ago following the request to have more complete and integrated modelling framework for both water supply and waste water systems.
The internal policy made collaboration between the two departments more time consuming; the overhead cost was charged in the case of using resources across departments.
Though the software was sold as a product series, the main business was to provide consultancy services to companies, such as spatial planning agencies.

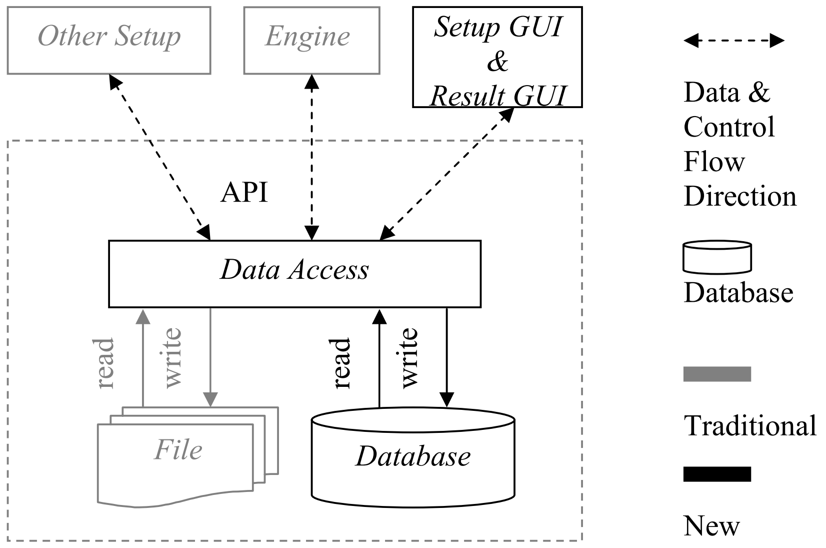


Figure 2. Run-time architecture of new data access module

Use Context: DHI's software applications are based on Windows operating systems because end-users were familiar with it. Most of the end-users have BSc. or MSc. in hydraulic engineering and are able to program by themselves; many of them used the DHI's applications during their studies.
The development of hydraulic modelling through the consultancy projects drove the development of the different software product lines. Many internal consultants feed their development of new modelling elements into the mainstream development branch of the respective department. They also put forward new requirements and report errors to the developers.

Software Engineering Organisation: The WSD and URBAN departments both had their own software development teams and project consultants. The WSD department was responsible for MIKE 11, which basically solves river network issues. The URBAN department was responsible for MOUSE and MU, which works in the sewer network domain. The development teams organised software to be in line with their respective application domains. The coordination between the two teams within each domain was easily done; however, since both MIKE 11 and MOUSE branch off from System 11, the growth of the organisation is not optimised because of double implementation in software development, for example assigning developers to solve the same task, spending unnecessary resources on development and maintenance.

Software Engineering Practice: Group of the developers have educational backgrounds on computer science and hydraulic engineering. *Engine* developers are hydraulic engineers while *GUI*s developers are software engineers/programmers. Many of the employees working today with software development began as consultants using and customising the software systems.
When System 11 was first developed, its architectural representation was not documented; only "live" document, that is the tacit knowledge of an experienced developer, was available. Even today, the developers understand the architecture of the system from source code and data format; it takes approximate 6 months for a new employee to understand the organisation of the code. The architecture is only temporary explicit because it is rarely documented on paper or file but mainly discussed at a whiteboard. In addition, the development organisation paid less attention to written documents because of its rather agile developments practice since it developed to accommodate the frequent requirements of the consultants of the same department.
A consequence of lacking a documented software architecture was the rise of redundant tasks. Because only reading the source code did not provide sufficient understanding for some of the new developers and deadlines were pressing, they implemented parts of the new code by the copy-and-paste technique. Another example is caused by the lack of a distinct outline for a *GUI*; we found some intensive computation functions in the *Setup part*, which we believed that it could be done in the *Engine* or a separated module.
Another necessity is the use of a file as a solution for coordinating asynchronous work between *GUI*s and *Engine* teams.

Technical Infrastructure: Nightly builds and the design of test suites for regression tests for the *Engine*s were used for enabling the agile development practice and at the same time keep a stable and high-quality product. New tests and setups were collected from the developers at regular meetings. A major effort was keeping different branches of these tests up to date when there were three applications: MIKE 11, MOUSE, MU, and two *Engine*s: one for MIKE 11, another for MOUSE and MIKE Urban, which both originated from System 11.

Technical Selection: Most of the hydraulic engineers were familiarised with C, C++, Delphi, or Fortran programming languages, therefore applications were implemented in these languages. In order to support safer dynamic memory allocation, technology benchmarking, communication to the other components in .NET environment and continuation for recruiting a new developer, they decided to change the programming language for implementation of the *Engine* from Delphi to be one of Fortran, unmanaged C++, managed C++, or C#. After the pros and cons discussion in the context of DHI, C# should be used.
Each department has its own data access pattern; MIKE 11 defined data structure in an XML-like proprietary format and stores the data in a *file* while MOUSE and MU also use a database.

3.2. Influences on the re-engineering project

Likewise we demonstrate influence in the re-engineering project by first introducing the design of new data access module and then discussing a subset of the categories from the first subsection: technical selection, software engineering practice, software engineering organisation, and use context.

Design of the New Data Access Modules: The long-term ambition of the re-engineering project is to have one *Engine* and one *Data Access* module for one-dimensional model simulation. The first sub-project is to handle data input and output, that means designing a new data access module. The new data access module is illustrated in Fig. 2. When *Setup GUI*, *Result GUI*, *Other Setup* or *Engine* request data via the *Data Access* module, they access all data through an application program interface (API) and respond to the request. The module provides a common data model and shared functionalities, e.g. read from/write to *file* or *database* for MIKE 11, MOUSE and MU. With the new design, the data model and data access are decoupled from the *GUI*s and *Engine*, which reduces maintenance effort and enables the application to be changed or more readily adaptable to change in the data model. Besides the implementation of a *Data Access* module, the change requires to free the *Engine* of the data access code.

Technical Selection: During the evaluation workshop, a developer commented on the implementation languages.
"*... With the new structure, our developer has to learn 5 different languages, ..., that will be a problem if they just want to try out something new.*" Besides the programming languages, the *Engine*, and the *GUI*s are implemented in, developers have also to understand the database management language, i.e. SQL[1], the language of a common data model, i.e. XML[2], and the API.

Software Engineering Practice: A prototype of the new *Engine* is available but its architecture has not been documented explicitly. When experienced developers resigned, "live" documentation left with them. The remaining developers encounter difficulties when continuing work on existing code structures. Also, the lack of documentation makes it difficult to coordinate substantial changes across the two departments.

Software Engineering Organisation: The software engineering organisational influences on the ability to implement the proposed architecture as a common product line became visible in the negotiation around the funding of such a joint project. Initially, the re-engineering project belonged to WSD. Thus co-ordination with the URBAN department seemed impossible. When URBAN had a meeting on joining the re-engineering project with WSD, URBAN delegates questioned about the resource allocation and the approval of budget for the joint project. After the meeting, we interviewed those delegates separately and the concept of "we and they" between the two departments were obvious. A WSD delegate guessed that the collaboration would be possible after WSD delivered an output. An URBAN delegate was eager to join because of willing to reduce a gap of development between these 2 departments. In early 2007, the

[1] Structured Query Language (SQL)

[2] Extensible Markup Language (XML)

internal reorganisation of the software development and consultancy postponed the approval of the budget for WSD and URBAN, and re-scheduled their joining into the re-engineering project.

Use Context: The technical selection would also hinder the difficulties for internal consultants to experiment as freely with the software as today. How would that influence the consultants' business and the flow from learning in the hydraulic modelling domain into the software development? This question was raised in the evaluation meeting when discussing the new design.

3.3. Summing-up

MIKE 11 is an end user customised software product that used for simulating one dimensional flow of river network. When we began with the MIKE 11 re-engineering project, we tried to understand the existing architecture and to institutionalise PLA. We found that the organisation of software development including the organisation history played a role. In addition, the business context, the use-situation, the work practices, the technical infrastructure and the technical selection played a part. Moreover, changing the design and development had implication for the whole simulation setup.

4. Discussions

This section describes several methods for producing PLA, presents organising for software product lines, and illustrates two successful case studies on organising for software product lines with comparisons to our work.

4.1. Methods for producing product line architecture

In order to produce software architecture that will satisfy the need of the product line in general and the individual products in particular, several methods are proposed, e.g. Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products (COPA), Family-Oriented Abstraction Specification, and Translation process (FAST), Feature-Oriented Domain Analysis

(FORM), Komponentenbasierte Anwendungsentwicklung (KobrA) which is German for “component-based application development”, and Quality-driven Architecture Design and quality Analysis (QADA). Matinlassi [13] compares these PLA methods. The initial processes of these methods rarely overlap. COPA first analyzes the customer needs. FAST, on the other hand, focuses on three processes, i.e. domain qualification, domain engineering and application engineering. FORM method starts with feature modelling to discover, understand, and capture commonalities and variability of a product line. KobrA uses the elicitation of user requirements within the scope of a framework as initial process. QADA first collects the driving ideas of the system and the technical properties on which the system is to be design.
Obviously, these methods start with the assessment of the current state of the software. Through the methods are not starting a focus on technical embedded systems, the majority of the reported cases are from that domain. Our case study indicates that one should in parallel start with the assessment of the business, the use-situation, the history of the organisational structure, and the work practices around the software development.

4.2. Organising for software product lines

Bosch [4] presents four organisational models for software product lines: development department, business units, domain engineering units and hierarchical domain engineering units. Moreover, he discussed the applicability of the model, advantages and disadvantages, case studies, and a number of factors that influence the organisational models. Adaptability of these modes is not explored. Böckle et. al. [14] describes in their work how to create an adoption plan and how to institutionalize product line engineering in an organisation. The structure of the adoption plan has three major parts: *i*) characterisation of current state, *ii*) characterisation of desired state, and *iii*) strategies, objectives, and activities to get from the current to the desired state. Even here, the history of organisation structure is not included in the assessment topics which we explicitly present in our work. In our case, one could consider evaluating the consequences of introducing a product line approach on the whole organisation when developing the adoption plan. Our analysis categories indicate the areas to be considered in such an evaluation. One example from our case: even such a small change as the introduction of the new design of the data access, which is clearly beneficial from a maintenance point of view, could hamper the traditional collaboration between software development and consultancy departments. That in turn might hinder innovation of the base software and might push the consultancy departments to maintain their own code base parallel to the software development organisation.

4.3. Software product lines: successful case studies

CelsiusTech Systems AB of Sweden had an experience of organisation in software product line [2]. The company builds large, complex, embedded, real-time shipboard command-and-control systems as a product line, based on a set of core software and organisational assets. CelsiusTech had to change its software, organisational and process structures to redirect the company toward a product line approach that yielded substantial economic and market benefit to the company. Dager [3] reveals the importance of organisational structure issues based on the software product line experience at Cummins Engine Inc., the world's largest manufacturer of commercial large diesel engines. The company changed their IT strategy to have a separate core asset group providing core assets to the product building groups.
Our experience of introducing PLA to an interactive system indicates that the different organisational and business factors are mutually dependent. When developing a PLA and perhaps adapting the organisation to support such an approach, a complex interplay of implications has to be taken into account.

5. Conclusions and future work

While we were participating in re-engineering the MIKE 11 project at the DHI Water Environment Health (DHI), we confirmed that producing product line architecture (PLA) is not only a matter of analysing commonality and variability, choosing a technical platform and infrastructure from an architectural point of view, but one must also consider the organisation and practice of software development as well as the business domain and use context. Our observation based on producing PLA for the software products that can be customised by end users rather than technical embedded systems.

Our analysis in Section 3 provides an empirical case showing how these organisational matters influence the design of common PLA. We elaborate the influences in relation to a number of contexts, i.e. business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection, which we support with citations from our field material. We compare our work with the other software product lines case studies, methods for producing PLA, and organising for software product lines.

At the time of writing, the organisation is restructuring; the cooperation between a consultancy organisation and the software development is changing. This created interest for our future work. because the experiment with a new architecture easily done in the old organisation might become more difficult for the new organisation as it would be regarded as influential for other redesign projects. The new organisation is decided upon the need to strengthen software development process and the design. DHI is implementing procedures to maintain collaboration and relationship between development and consultancy departments without hampering the work processes and development logistics required by the new organisation structure.

References

[1] D. Ganesan, D. Muthig, J. Knodel, & K. Yoshimura, Discovering organizational aspects from the source code history log during the product line planning phase—a case study. *Proc. 13th Working Conf. on Reverse Engineering*, Benevento, Italy, 2006, 211-220.

[2] L. Brownsword & P. Clements, A case Study in Successful Product Line Development. *Technical Report*, CMU/SEI-96-TR-016.

[3] J. Dager, Cummin's Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls, *Software Product Line: Experience and Practice* (MA: Kluwer Academic Publisher, 2000).

[4] J. Bosch, *Design and Use of Software Architectures: Adopting and evolving a product-line approach* (Addison-Wesley, 2000).

[5] M. E. Conway, How do committees invent?, *Datamation*, *14*(4), 1968, 28-31.

[6] A. Huczynski & D. Buchanan, *Organizational behaviour: an introductory text* (Prentice-Hall, 1991).

[7] P. Clements & L. Northrop, *Software product lines* (MA: Addison-Wesley, 2001).

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

[9] H. Muccini & A. van der Hoek, Towards Testing Product Line Architectures. Electronic notes, *Theoretical Computer Science*, *82*(6), 2003.

[10] DHI Water Environment Health website, **http://www.dhigroup.com**, last visited 28-08-2007.

[11] ESRI GIS and Mapping Software website, **http://www.esri.com/products.html#arcgis**, last visited 28-08-2007.

[12] Y. Dittrich, K. Rönkkö, J. Erikson C. Hansson & O. Lindeberg, Co-Operative Method Development: Combining qualitative empirical research with method, technical and process improvement, *accepted for publication in Empirical Software Engineering Journal*.

[13] M. Matinlassi, Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. *Proc. 26th*

International Conf. on Software Engineering, Scotland, UK, 2004, 127-136.

[14] G. Böckle, J. Muñoz, P. Knauber, C. W. Krueger, J. C. Sampaio do Prado Leite, F. van der Linden, L. Northrop, M. Stark & D. M. Weiss, Adopting and institutionalizing a product line culture. LNCS 2379, *Proc. 2nd International Conf. on Software Product Lines*, San Diego, CA, USA, 2002, 49-59.

TAKING CARE OF COOPERATION WHEN EVOLVING SOCIALLY EMBEDDED SYSTEMS: THE PLONEMEETING CASE

Hataichanok Unphon, Yvonne Dittrich
Software Development Group
IT University of Copenhagen
Denmark
{unphon, ydi}@itu.dk

Arnaud Hubaux
PReCISE Research Centre
Faculty of Computer Science
University of Namur
Belgium
ahu@info.fundp.ac.be

ABSTRACT. This paper proposes a framework to (i) analyse the contexts of socially embedded systems and (ii) support the understanding of change during their evolutions. Our finding is based on a co-operative project with a government agency developing a partially-automated variability configurator for an open source software product family. By employing our framework, we realised that the way variations and their management are implemented have to accommodate work practices from the use context as well as development practice, and here especially the cooperation within the development team and between users and developers. The empirical evidence has confirmed our understanding of what is relevant when estimating the evolvability of socially embedded systems. We propose to use our framework in architecture-level design and evaluation in order to take these cooperative relationships into account early in the evolution cycle.

1. Introduction

In this paper, we study the effects of change on the design of software from the socio-technical perspective. Our objectives are to (1) clarify the meaning of socially embedded systems, and (2) understand how to support their further development. More specifically, we are interested in understanding how developers can evaluate the impact of changes on the cooperation within the team and between developers and users. We report here on a study of an open source project developed by a Belgian government agency. The case study has confirmed our prior understanding of what dimensions of categories to investigate. A model of contextual categories relevant to assess design proposals developed in [21] is successfully applied to the case. The study confirms that the cooperation between developers and between developers and users have to be considered when evolving the architecture of a software product. The analysis using the framework provided us with a better understanding of the project and helped us to improve and complement the development process with state-of-the-art techniques. We therefore recommend using the model as a tool for structuring design discussions when drafting architecture and when evaluating evolutions for socially embedded systems.

This paper is outlined as follows. Section 2 introduces terms and definitions. Section 3 presents case description. Section 4 explains the research approach. Section 5 illustrates evolution of the PloneMeeting. Section 6 is discussion. Section 7 is conclusion and plan for future works.

2. Terms and definitions

This section defines two terms: (1) socially embedded systems, and (2) software evolvability, as shown in sections 2.1 and 2.2 respectively. To avoid terminological confusions between system and software, they are used interchangeably.

2.1. Socially embedded systems

An embedded system has ongoing interaction with a dynamic external world [15]. The concept of embedded systems is widely known in terms of a combination of computer hardware and software, and potentially additional mechanical or other parts, designed to perform a dedicated function. Wolf [22] defined the term embedded systems as 'any computer that is a component in a larger system and that relies on its own microprocessor'. In this paper, we call such systems *technically embedded systems*. Typical examples are MP3 players, telephone switches, and hybrid cars. Usually, design decisions are mainly constrained by interfaces to hardware or mechanical specifications. We define the term *socially embedded systems* as *any system that can be modelled intensively according to the environment and practices of its end-users*. ERP systems, e-government applications, virtual office software, and decision support systems are examples of socially embedded systems. Design decisions of socially embedded systems underline the importance of the human interaction with and cooperation via the systems for the societal activities. Lehman [18] has defined a characteristic of *E*mbedded programs (*E*-programs) that it has become a part of the world which it models. This implies a constant pressure for change. Since focused human or societal activities, the usability of the system is the main concern of *E*-programs. The close co-operation between end-users, people working with the systems on a daily basis, and developers throughout the entire development process is strongly recommended for capturing the contexts and qualities of use that cannot be fully anticipated at the initial phase.

Participatory Design (PD) is a research discourse investigating in how to support cooperation between users and developers when designing socially embedded systems [17]. The focus of *PD* is to let IT-designers work directly with their end-users in the end-user's own environment and come up with design ideas in real-life work situations. Floyd et al. [12] have proposed *STEPS*, *S*oftware *T*echnology for *E*volutionary *P*articipative *S*ystem Development (*STEPS*) in order to promote user-developer cooperation as the use context is considered an inherent a part of the development. In this approach, software development does not start from pre-defined problems, but must be considered as a learning process involving the unfolding of the problems as well as the elaboration of a solution tackling the problems.

Dittrich et al. [9] elaborated on how co-operation with the users can take place in an industrial project without jeopardising the planning and control of the development process. By employing mock-ups, prototypes, or scenarios of use, end-users can experience the new technology, and IT-designers can experience the new work practice.

Socially embedded systems often allow users to tailor the software to specific needs. Examples of end-user tailoring categories are customisation, composition, expansion, and extension [11]. Apart from tailoring, the socially embedded systems also have to evolve over time.

2.2. Software evolvability

Belady and Lehman [3] first introduced and used the term *evolution* as 'a sequence of changes to the system over its lifetime which encompassed both development and maintenance'. Cook et al. [4] has developed the concept of software evolvability based on maintainability characteristics in ISO 9126, and proposed the evolvability measurement at different levels of abstraction, i.e., at pre-design, architectural, detailed design and source code levels. The concept of evolvability brings together factors from three main areas: (1) software product quality, (2) software evolution processes, and (3) the organizational environment in which the software is used. They have defined the concept of software evolvability as 'the capability of software products to be evolved to continue to serve their customer in a cost-effective way'.

To survive in today's competitive software market, it would be too restrictive to limit software evolvability to maintenance issues only. The growth dynamics of a system depends highly on the business context. To increase a market share, e.g., it may be vital to bring out new features. Yet, a system that is used will be changed [19]. We here define *software evolvability* as *the adaptability of software in order to serve the needs of use and business contexts over time reflecting on its architecture*. Software architecture represents a common abstraction of a system that many of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus, and communication [2]. When the needs of use and business contexts trigger changes to the software, the stakeholders must handle the needs based on the architecture.

In this paper, we address how to relate cooperation among developers and between users and developers when evolving socially embedded systems. Based on previous research [21], we recommend framework that builds up on six contexts: business, use, software engineering organisation, software engineering practice, technical infrastructure, and technical selection. The *business context* is the context or environment to which the system belongs. The *use context* relates the system to the work practices of the intended users. The *software engineering organisation* is the organisational context in which the software development is carried out. The *software engineering practice* refers to the analysis of the work practices of the system developers. The *technical infrastructure* lists the hardware and basic software assets backing the system. The *technical selection* is part of a suggested design and needs to be seen in the context of existing and planned systems, as well as in the context of other systems that are part of the same design.

Others have used the notion of context or contextual factors before. Kensing [16] has proposed a conceptual framework that IT-designers should be aware of when they design IT-applications to meet the needs of a specific organisation. The framework addresses: (1) project context, separating into design context and implementation context; (2) use context, dealing with work practice context and strategic context; and (3) technical context, interacting with system context and platform context. Kensing does not apply the framework to reflect on concrete design proposals. Dittrich and Lindeberg [8] developed Kensing's framework further by mapping out contextual factors in order to understand the suitability of a – from a technical viewpoint – less advanced design for a specific industrial setting. Here, three contextual dimensions are used: development, use, and technical. We developed this framework to support architecture-based analysis when planning to evolve software products.

3. Case description

EASI-WAL[1] is the Belgian government agency in charge of (1) the simplification of administrative tasks and (2) the global

[1] The EASI-WAL website, http://easi.wallonie.be/.

computerisation of the administrative processes of the Walloon region[2]. To this end, EASI-WAL develops open source web-based applications assisting the Walloon public institutions. In order to stay independent of external contractors and to pool the efforts, the developers set out to develop applications generic enough to deal with the difference in scales and behavioural disparities of the various institutions.

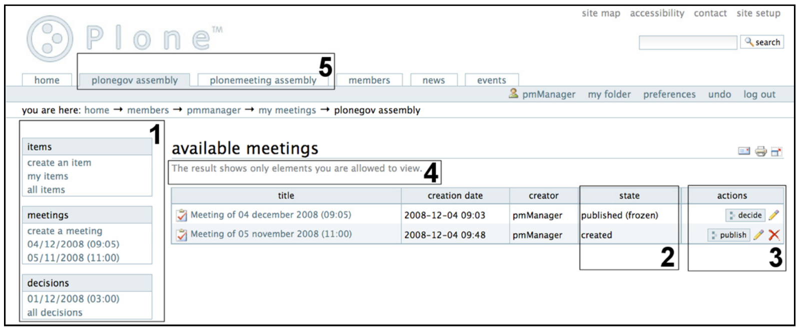
For each of their products, the developers struggled to properly elicit and express the requirements. Although, the elicitation of the requirements followed an iterative process of prototype demonstration and validation with users together with impromptu interviews with the users and results of developers' experience in the field, it did not follow a specific elicitation protocol. As we examined their case, it turned out that most of the challenges came from the identification of the common parts, shared by all the institutions, and the specific parts, peculiar to a single or a set of them. We thus advocated software product line engineering [20] as solution to their problem of systematising the engineering of product families. In order to assess the potential of software product line engineering, the developers agreed to apply it to an existing application, *viz*. PloneMeeting [7, 14].

PloneMeeting provides advanced meeting management functionalities like meeting workflow specifications and document generation. Figure 1 shows a screenshot of the graphical user interface of PloneMeeting. To further illustrate the functionalities proposed by PloneMeeting, we trace three representative use cases: meeting management, meeting workflow management, and document generation.

Meeting management usually follows a three-step process. First, the meeting items are created and validated. Secondly, a meeting is created and existing meeting items are added to the meeting agenda. Third, after publication, the meeting takes place and the decisions related to the meeting items are recorded (label 1 in Figure 1, a.k.a. Figure1.1).

Meeting workflow management. Central to PloneMeeting is the concept of meeting itself. Each meeting is associated to a state of a workflow. The meeting states evolve according to a pre-defined workflow designed and selected at the installation of PloneMeeting. A

[2] The Walloon region gathers a third of the Belgian population and covers about a half of the territory. The region includes a French-speaking and a minor German-speaking community.



you are here: home → members → pmmanager → my meetings → plonegov assembly

available meetings

The result shows only elements you are allowed to view.

title	creation date	creator	state	actions
Meeting of 04 december 2008 (09:05)	2008-12-04 09:03	pmManager	published (frozen)	decide
Meeting of 05 november 2008 (11:00)	2008-12-04 09:48	pmManager	created	publish

Figure 1. The PloneMeeting graphic user interface

typical workflow contains states like *Created*, *Published, Frozen, Decided*, *Closed and Archived* (Figure 1.2). The respective next states can be seen in the *actions* column (Figure 1.3). On top of specifying valid states, a workflow contains guards and potential actions on the transitions. Guards basically check the permissions of the user willing to trigger the transition (Figure 1.4). The available workflows are selected during the PloneMeeting base configuration. The user selects one of these workflows for each meeting created. The portal tabs show two meeting configurations, i.e. *plonegov assembly* and *plonemeeting assembly* (Figure 1.5).

Document generation. Document generation is an essential functionality that must be provided by a meeting management system dedicated to governmental administration. For that reason, every meeting item, meeting and decision can be exported into various formats like PDF, ODT and DOC in a single click. Different document templates can be specified and selected in the PloneMeeting configuration menu.

4. Research approach

Our research approach is a combination of co-operative method development (CMD) [10] and grounded theory approach [13]. The CMD is a domain-specific adaptation of action research consisting of three evolutional phases: (1) understanding practice, (2) deliberate improvements, and (3) implement and observe improvements. The grounded theory approach is an explicit and systematic technique for developing theory iteratively from qualitative data. The initial analysis of data begins without any preconceived categories. When an interesting pattern emerges, it is repeatedly compared with existing data, and additional data is collected to support or refute the emerging theory. We have applied the grounded theory approach to our observation and field notes, transcripts of interviews and workshops, and other material collected through these activities including relevant literatures.

We have applied the CMD to the basis of our fieldwork research aiming at institutionalising variability management for the PloneMeeting product family. Initially, we conducted interviews with three PloneMeeting developers, one working for EASI-WAL and the two others working for two different city councils. We studied functionality of the existing application, structure of the source code, and checked out the major related product, i.e. PloneTask. Besides these contact meetings, we held two workshops in which we successively studied: (1) the product line architecture of PloneMeeting, and (2) the feature modelling and configuration tools. In the first workshop, the main responsible developer of PloneMeeting reflected and explained the current architecture in terms of implementation techniques and rationales behind it. In the second workshop, we presented a number of mainstream variability management tools on the market and suggested a work plan for integrating an automated configuration tool into the existing PloneMeeting architecture. The development of the configurator is currently supervised by a group of researchers from the PReCISE research centre[3], University of Namur, Belgium.

5. Evolution of PloneMeeting

This section reveals the evolutionary story of PloneMeeting into three episodes: (1) Collège: the origin of PloneMeeting, (2) the PloneMeeting product family today, and (3) towards the PloneMeeting configuration wizards, as shown in sections 5.1, 5.2, and 5.3 respectively. For each of these episodes, we will look into the contexts presented in section 2.2.

[3] The PReCISE website, http://www.fundp.ac.be/universite/ interfacultaire/precise/.

5.1. Collège: the origin of PloneMeeting

Business context. Collège was a web-based application dedicated to the management of the official meeting of municipal authorities in Belgium. Developed in 2004, Collège has been in production three years. Part of the Belgium policy for e-government is to use open source software as much as possible.
Technical infrastructure. **Collège is a monolithic program or a self-contained program[4].**
The **technical selection** is not available.
Software engineering practice. The developers report that the evolution of Collège was a challenge. Due to a lack of explicit architecture, most modifications were done at the source code level. Implementing a simple meeting template took a day. Templates could not be shared easily.
Software engineering organisation. The Collège developer team was located in the same region. When a city requested a Collège product, the team copied the existing Collège source code and modified it according to the city's specific needs. In case of updates, a power-user, who acted as administrator, had to go through the whole code base and manually update the code and resolve the conflicts.
Use context. Laws and legal regulations constrained use cases. For example, Collège can generate a document only in PDF format because an official document should not be modified.

It seemed impossible for users to change anything apart from three or four configurable elements. Most of the changes involved programming. During the first workshop on the PloneMeeting architecture, a developer mentioned that "*Collège was used in three cities. …Three cities are OK, but what if you have 45 cities? This way of doing things is not manageable. This product was not designed correctly for a lot of organisations. That is the first limitation of the old architecture.* ".

[4] Monolithic program or self-contained program indicates a program which is contained in a single function of the large program.

5.2. The PloneMeeting product family today

Business context. Since June 2007, PloneMeeting has been totally refactored from Collège and has gathered interest from other government agencies. PloneMeeting has been tailored for four main organisation types, i.e. general purpose, city, government, and parliament. A version for the parliament has been planned.

Use context. PloneMeeting is used by civil servants. The amount of users per product is approximately 100 with up to 10 different roles like meeting manager or reviewer. The products are customised. Changing the display language and document layout are examples for simple customisations.

When installing PloneMeeting power-users also can select their own workflows based on pre-defined UML state charts available in the PloneMeeting set of core assets. A power-user is free to implement its own workflow but it requires advanced programming skills.

Technical infrastructure. PloneMeeting is developed on the top of an open source technology stack: Plone[5], Zope[6], and Python[7].

Technical selection. Since PloneMeeting is built on top of Plone, some architectural dependencies are directly managed by Plone. Using a plug-in architecture, the core of PloneMeeting is relatively simple and flexible enough so that nearly every aspect of its input and output can be modified by the plug-in. This mechanism is used for customising the behaviours of PloneMeeting.

The most sensible part of the PloneMeeting architecture is the workflow. A workflow is composed of several states, actions, and guards and involves several categories of users. PloneMeeting comes with several workflows so as to meet the work practices of most organisations. PloneMeeting employs ArchGenXML to generate Python code from the workflows designed in UML before the product configuration. Although ArchGenXML builds on architecture-centric, model-based and test-driven development, PloneMeeting developers still have to write the code e.g. the body of a method or the trigger of a workflow transition manually.

[5] The Plone website, http://plone.org.

[6] The Zope website, http://www.zope.org.

[7] The Python website, http://www.python.org.

The current configuration tools do not enforce any constraints, e.g. a power-user can choose more than one option for a feature even if they exclude each other.

Software engineering organisation. PloneMeeting belongs to larger open source project, namely PloneGov[8]. Five developers belonging to different municipalities or organisations are working part time on PloneMeeting. The developers gather requirements from their own organisations/entities. The architectural change occurs after developers get feedback from end-users.

One of the developers has the main and coordinating responsibility. He reviews the modifications, checks in the updated source code, and tests it. To develop a subsequent product, every change is done after a careful discussion with all the developers.

PloneMeeting developers do not only provide the software products to open source communities, but also are power-users and administrators for their own organisations. To reduce development time, the vision is to simplify configuration and leave it to power-users who have no development charges. When we discussed the configuration of the product, we were puzzled by the fact that configuration choices can be performed at different times in the lifecycle. This absence of control would notably allow configurations incompatible with the current state of the system (e.g. unknown workflow states for running meeting objects). We thus started to differentiate configuration according to the binding time and evolvability of the choices – some can only be done once when installing the software, others can be changed repeatedly as part of the usage. Still, categorisation needs to be clarified in a systematic way. One approach would be to employ the categories used in end-user tailoring: customisation, composition, expansion, and extension, instead of the term configuration.

The **software engineering practice** looks different depending on the role in the development process. We interviewed a developer working for an organisation that wants to replace their old systems with PloneMeeting. In the beginning, the developer met end-users of the old systems every second week. The developer presented a prototype of PloneMeeting and the end-users commented on it. He recorded the requirements and change requests to the Trac System, an enhanced

[8] The PloneGov website, http://www.plonegov.org

Table 1. Plan for PloneMeeting wizards and configuration tools

KIND OF CONFIG.	Initial install	Product line related change	Simple parameter change
FREQ. OF USE	Once	Once a year	Once a week
ACTOR	Power-user with business knowledge	Power-user with business knowledge	Power-user with technical knowledge
INPUT/ PRE-COND.	• Business analysis	• Business analysis • PloneMeeting plug-in to Plone	Simple parameter change request
OUTPUT/ POST-COND.	• PloneMeeting plug-in to Plone • BuildOut	• Adapted PloneMeeting plug-in • Adapted BuildOut • Migration script (if big change)	Configuration data change
COMPONENT TO USE	Wizard	Wizard	Configuration tool

Wiki and issue tracking system. Then he changed the configurations, implemented and released the necessary code changes, which took about 2 weeks.

PloneMeeting developers join in weekly *sprints*. A sprint is a focused development session lasting anywhere from a day to a week. During sprints, the developers prioritise the requirements, design, code, test and at the end of a day release a new version of PloneMeeting. Criteria for giving high priority requirements are (1) a requirement is highly required from several organisations, and (2) a requirement comes from the organisation that allocates the most resources, e.g. time, to the project. Most of the requirements are shared among organisations.

Developers are located very close to the user organisations for which they create the local installation. For this, they have to configure and adapt a PloneMeeting profile manually, which is a time consuming task considering the great deal of technologies involved. To generate a skeleton of a product, they reuse and adapt, if needs be, the existing UML models.

As much of the development is model driven, most of the code is generated. The architecture is not explicitly documented. The parallel development provides a challenge for the model driven approach as changes have to be coordinated at the model level as well as at the source code level. Since the deployment is dynamic, modification of the core product are automatically deployed on every new installation of PloneMeeting. In case of major changes that affect the data structures, the developers provide a script for data migration so that local versions can be upgraded.

Another time-consuming task for developers is to handle change requests after base installation. An example is the archive meeting case. A stakeholder requested to archive documentation of meetings. In order to add this unanticipated change request, the developer needed to change a number of parameters, customise and add some status to the workflow, write some Python method, etc. From the developers' point of view, a critical issue in the shared development environment is an architecture that allows developers to capture commonality of the organisations into common parts as well as to easily integrate specificity among organisations that share and use PloneMeeting.

5.3. Towards the PloneMeeting wizards

A practical outcome of our research co-operation and the facts revealed with our framework is the development of an automated configurator for PloneMeeting. During the second workshop on feature modelling and configuration tools, we came up with a plan for the PloneMeeting wizard and configuration tool. Table 1 summarised the ideas and will be detailed later in this section. The difficulties described above indicate the need to better structure the configuration tasks. Different actors who configure and customise different aspects of the software at different points in time need different kinds of support. From the introduction of the development, business context and software engineering organisation, and technical infrastructure remain the same, but the use context, software engineering practice, and technology selection are impacted.

Use context. PloneMeeting has at least 15 power-users from different government agencies in Belgium. PloneMeeting developers have an ambition to support their power-users to configure bigger parts of the software and be as independent from developers as possible. Although the power-users have little knowledge of software engineering, the wizards and the configuration tools can enable them to express their requirements by choosing from a list of existing features. For instance, to implement a meeting archive feature, they would use a wizard. The wizard would generate an instance of PloneMeeting product family using a given workflow including the meeting archive.

However, different levels and frequencies of configuration have to be distinguished. When first installing the software, a power-user with business knowledge will configure the basic family member using one wizard. The wizard will generate some components. The subsequent stages represent modification of the base configuration. The tools supporting the latter stages constrain configurations to what is allowed with respect to the running configuration.

Software engineering practice. In order to create a wizard, developers are eliciting the user-level variability that will be used for developing the PloneMeeting wizard and configuration tool. This elicitation will be done at the requirements level. However, the requirements are not completely documented. Often, requirements gathering have been informal and chaotic. PloneMeeting has been refactored from Collège whose developer did neither use the Trac system nor any other tool for documentation. Furthermore, the documentation of requirements and product analysis are not standardised. PloneMeeting developers now have an ambition to have robust processes for requirement elicitation and traceability. As a starting point for the development of the PloneMeeting wizard and the configuration tool, we suggested to list all the questions that end-users, power-users and developers should be asked in parallel with likely answers, in order to systemise their elicitation process.

Technical selection. Developers have distinguished configuration elements into features and simpler parameters. Email of power-user and meeting type are examples of a simple parameter. Document generation, document template, meeting archive and meeting workflow are examples of features. The simple parameter is used in settings that neither influence any feature nor effect on the rest of the configuration. The features are categorised into features that can be changed during runtime, e.g. the interface language, and features that are decided when first installing the product, e.g. the meeting workflow. If features need to be changed, the configured PloneMeeting instance must be removed and replaced by an updated version. There are thus three levels of configuration, which can change at different frequencies and require different expertises.

The currently used feature modelling language is the cardinality-based feature diagram of Czarnecki et al. [6]. The configuration process under evaluation is the multi-level staged configuration process, which offers a fine-grained and well-defined decomposition of the modelling perspectives into linked feature diagrams organised in sequential levels. The sustainability and gain of this approach are still to be diligently demonstrated.

In order to automate the build process, developers choose BuildOut, a tool for developing, packaging, and deploying Plone applications. Another design option that will facilitate maintenance would be to enhance the existing PloneMeeting configurator with automation and export/import facilities. At the time of writing, the PloneMeeting developers have decided to represent the feature diagram directly in Python in order to avoid any dependency to existing tools and ease the binding with the code.

6. Discussion

In the discussion we take up three points: (1) the use of the six contextual dimensions as a framework to prepare and evaluate changes to software products, (2) how to introduce the framework into the architecture evaluation practice in a company, and (3) how product line and product line configuration approaches that are developed for technically embedded systems can be applied to socially embedded systems.

Using the framework to design and evaluation of changes. The PloneMeeting case suggests that the six contextual dimensions – business, use, software engineering organisation, software engineering practice, technical infrastructure, and technical selection – can be applied in order to understand the effects of change for the socio-technical context of socially embedded systems. The six contexts helped us to map out aspects affected by changes which have to be considered and resolved when analysing specific requirements.

The analysis provided in this article helped to clarify the complexity of the configuration task for the PloneMeeting product family. When evaluating the concrete design for the configuration mechanism, the framework can be used to analyse and evaluate the impact of the chosen solution on the cooperation, respectively, the distribution of tasks between the local developers, the power-users, and end-users.

Introducing the framework into architecture evaluation practices. The introduction of a method into an existing software development organisation is not always a straightforward task. Ali Babar et al. [1]

have presented an empirical investigation of factors influencing industrial architecture evaluation practices. They have categorised the findings into: (1) organisational factors, involving: engagement models, governance frameworks, supporting software engineering organisational structures, design decision documentations, funding models, and training; (2) technical factors, including: quality attributes being evaluated, challenges caused by integration issues, techniques and tools for representing and visualising architectures, types of evaluation required, and methods and guidelines used; (3) socio-political factors, relating to: soft skills, organisational politics, and vendor involvement; (4) managerial factors: pulling by management, support and commitment, objectives of evaluating architectures, and stakeholder-centric issues; and (5) business factors: resulting from business needs and industry standards, and requirements of business case. The CMD approach is based on cooperation between researchers and practitioners when deliberating, introducing and evaluating improvements and will thus provide a good base to explore the specific benefits and hinders when introducing our framework. So far, we did not explicitly introduce the framework into the development lifecycle. The analysis of its result helped us to better understand the context of PloneMeeting and worked as a catalyst to improve the development practices. This paper is only the first stage towards its understanding and uptake by practitioners. Additional work is still needed to assess the actual impact it will have once used by the different stakeholders.

How to apply product-line architecture approaches to socially embedded systems? In the existing work, feature modelling has been applied to technically embedded systems [5]. But PloneMeeting is considered as a socially embedded system because of the extensive interactions with the environment and practices with users. The design decisions are loosely constrained by static conditions, and the contexts and qualities of use cannot be fully anticipated in the starting phase. The next step is to explore how feature modelling can be applied, in general, to socially embedded systems and evaluate to what extent it enhances their usability.

7. Conclusion and future works

The PloneMeeting case demonstrates how our framework can be used to understand the impact of changes in socially embedded systems on the cooperation among developers and between users and developers. In the PloneMeeting case, introducing a wizard and a configuration tool turned to solve some problems in the local development by controlling the instantiation of PloneMeeting and its features. From the early phase of the development of the wizard and the configuration tool, the changes are designed to support the work and development practices of the power-users.

At the time of writing, we are assessing the impact of the wizard and configuration tool on the development and deployment of the application in production sites. We are polishing up the terms and definitions as well as the analytical tool proposed in Section 2 with different cases. Apart from that, we keep up with the questions and challenges posed in the discussion section.

References

[1] M. Ali Babar, L. Bass, and I. Gorton, "Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation", *in S. Overhage et al. (Eds.)*, *the 3rd Int. Conf. on the Quality of Software Architecture (QoSA)*, *LNCS 4880*, Spinger-Verlag, 2007, pp. 90-107.

[2] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd edition, Addison-Wesley, 2003.

[3] L.A. Belady, and M.M. Lehman, "A Model of Large Program Development", *IBM Systems Journal 15(1)*, 1976, pp. 225-252.

[4] S. Cook, H. Ji, and R. Harrison, "Software Evolution and Software Evolvability", *Working paper*, University of Reading, UK, 2000.

[5] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", *Proc. ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE'02)*, *LNCS 2487*, Springer-Verlag, Germany, 2002, pp. 156-172.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Through Specialization and Multi-level Configuration of Feature Models", *Software Process: Improvement and Practice 10(2)*, *Special issue on Software Product Lines*, John Wiley & Sons, 2005, pp. 143-169.

[7] G. Delannay, K. Mens, K., P. Heymans, P.-Y. Schobbens, and J.-M. Zeippen, "PloneGov as an Open Source Product Line", *Proc. 3rd Int. Workshops on Open Source Software and Product Lines*, 2007.

[8] Dittrich, Y., and O. Lindeberg, Designing for changing work and business practices, *in Adaptive evolutionary information systems*, IGI Publishing, USA, 2003, pp. 152-171.

[9] Y. Dittrich, and O. Lindeberg, "How Use–Oriented Development can Take Place", *Information and Software Technology 46(9)*, 1 July 2004, pp. 603-617.

[10] Y. Dittrich, K. Rönkkö, J. Erikson, C. Hansson and O. Lindeberg, "Co-Operative Method Development: Combining qualitative empirical research with method, technical and process improvement", *Empirical Software Engineering Journal 13(3)*, Kluwer Academic Publishers, 2008, pp. 231-260.

[11] Eriksson, J., *Supporting the Cooperative Design Process of End-User Tailoring*, Doctoral Dissertation, Department of Interaction and System Design, School of Engineering, Blekinge Institute of Technology, Sweden, 2008.
[12] C. Floyd, F.-M. Reisin, and G. Schmidt, "STEPS to Software Development with Users", *Proc. 2nd European Software Engineering Conf.*, *LNCS 387*, 1989, pp. 48-64.
[13] Glaser, B.G., and A. Strauss, *Discovering of Grounded Theory: Strategies for Qualitative Research*, Sociology Press, 1967.
[14] A. Hubaux, P. Heymans, and H. Unphon, "Separating Variability Concerns in a Product Line Re-Engineering Project", *Proc. 2008 AOSD workshop on Early Aspects*, Brussels, Belgium, 2008.
[15] Kaelbling, L.P., *Learning in Embedded Systems*, MIT Press, 1993.
[16] F. Kensing, "Participatory Design in a Commercial Context – a Conceptual Framework", *Proc. Participatory Design Conf.*, USA, 28 Nov.-1 Dec. 2000, pp. 116-126.
[17] Kensing, F., *Methods and Practices in Participatory Design*, ITU Press, Denmark, 2003.
[18] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", *IEEE 68(9)*, 1980, pp. 1060-1076.
[19] M.M. Lehman, "On Understanding Law, Evolution, and Conservation in the Large-Program Life Cycle", *Systems and Software 1(3)*, 1980, pp. 213-231.
[20] Pohl, K., G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, USA, 2005.
[21] H. Unphon, and Y. Dittrich, "Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture", *Proc. IASTED Int. Conf. on Software Engineering*, Innsbruck, Austria, 2008 , pp. 178-183.
[22] W. Wolf, "What is Embedded Computing?", *Computer 35(1)*, IEEE Computer Society, 2002, pp. 136-137.

MAKING USE OF ARCHITECTURE THROUGHOUT THE SOFTWARE LIFE CYCLE – HOW THE BUILD HIERARCHY CAN FACILITATE PRODUCT LINE DEVELOPMENT

Hataichanok Unphon
IT University of Copenhagen & DHI Water Environment Health
Denmark
unphon@itu.dk

ABSTRACT. This paper presents an empirical study of how the application of genuine architecture can be employed beyond the design phase of product line development. The study is based on a co-operative research project with a company developing product line architecture for hydraulic modelling software. By concretising the architecture as a build hierarchy the architecture mediates the evolution of the design throughout the whole software life cycle. The empirical evidence has confirmed the improvements of (1) the software quality and flexibility, (2) the communication and co-operation with new developers, (3) the distribution of work and parallel implementation, and (4) the foreseen usage by hydraulic and environmental consultants who tailor the software. Our research further indicates requirements for the architectural analysis tools that are deliberately embedded in the daily development practices.

1. Introduction

An ambition to establish architecture as the key aspect of software development prompts this work. The importance of software architecture has been recognised in the software engineering community for decades [3]. But, in many software industries, the use of explicit architecture is very limited. This paper presents empirical evidence of how developers can make use of architecture beyond the initial design phase. A case study shows that a team of software developers "concretises" architecture as a build hierarchy[1]. The developers use the build hierarchy to check the compliance of their source code against the architecture's structure when they build the software. The developers get constant feedback about the match between design architecture and code architecture. As a consequence, it improves (1) the software quality and flexibility, (2) the communication and cooperation with new team members, (3) the distribution of development tasks and parallel implementation, and (4) the usage by hydraulic and environmental consultants. However, there are some challenges to the build hierarchy and its consequence with respect to different aspects of evolvability which should be explored in further detail later. Instead of inventing a new insight, the contribution of this work emphasises on making use of the architectural knowledge throughout the software life cycle.

This paper is outlined as follows. Section 2 presents some challenges towards evolving architecture that is intentionally shared with a family of software products. Section 3 presents case description. Section 4 explains research approach. Section 5 illustrates concretisation of the architecture as the build hierarchy. Section 6 shows beneficent effects of build hierarchy. Section 7 presents challenges towards evolvability. Section 8 is discussion. Section 9 draws conclusion and requirements for the architectural analysis tools.

2. Evolving product line architecture

A software product line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product line architecture. In addition, the product line consists of the software products that are developed using mentioned reusable assets [6]. Software architectural design is one of the important incorporation elements in the software product line. When design architecture and code architecture are handled independently, the code architecture

[1] A build hierarchy is a technique to organise a series of generating executable code based on dependencies between components.

usually diverges from the design architecture [12]. The design architecture can be thought of as the ideal implementation structure. The design architecture can be described using layers. Layers are usually described using stacked rectangular boxes. Proximity between these boxes represents allowable interfaces between components in different layers. The code architecture describes how the source code, binaries, and libraries are organised in the development environment [19], and should be implemented in the design architecture. The problems, then, are (1) how to describe the compliance between the design architecture and the code architecture, and (2) how to maintain that throughout the development.

3. Case description

DHI Water Environment Health (DHI) is a pioneering organisation that develops software applications for hydraulic modelling [8]. In 1972, System 11 and System 21 were one of the first computational modelling systems developed at DHI to simulate water flow patterns with the help of one-dimensional and two-dimensional models. A three-dimensional simulation was developed in the 1980s. Originally, the organisation focused on hydraulic characteristics research, not on software engineering. Software development and software maintenance were challenges on only a small scale. In the late 1980s, DHI released the MIKE 11 and the MOUSE software products. Both products were originated from System 11 following the requests of different usages, i.e. open channels and pipe networks. The main users of these products are consultants that do simulations of hydraulic conditions, i.e. water level and flow, and analyse the hydrological effects of environmental change, so-called hydraulic and environmental consultants. Due to the different market needs the ownership was split into different consultancy departments and, in the last decades MIKE 11 and MOUSE have been developed and maintained in parallel. Released in 2005, MIKE URBAN followed the request to have a more complete and integrated modelling framework for both water supply and waste water systems.

After decades of successful use and development, the requirements of the software have evolved as well. In particular there is a growing tendency that the software is used in a more general setting e.g.

scripting and scheduled forecasts. The company was faced with the challenge of identifying and developing a kernel for data handling, simulation setup, and graphical interaction with simulations and their results. For this reason the idea of re-engineering the existing software products was initiated. The first re-engineering project started with the MIKE 11 engine[2]. Later on, the MOUSE engine was merged into the MIKE 11 re-engineering project. While the merged re-engineering project was under way, the organisation was undergoing change. DHI set up a software product department in order to strengthen the software development process and the design. The software product department has taken development activities and ownership of DHI's software products. As a consequence, the department decided to re-engineer the core computational parts of some of the one-dimensional simulation software products, i.e. MIKE 11, MOUSE and MIKE URBAN, in a project called MIKE 1D. The project is estimated for 360 person weeks of implementation.

4. Research approach

The cooperation with DHI addressed the introduction of product line architecture into product development. The basis for the research described here is the fieldwork which I have been involved in for two and a half years. I wrote a research diary documenting daily observations, interviews, and meetings. As a field worker, I was expected not only to observe, but also to influence the projects in which I participated. The research was designed as action research by following the cooperative method development approach (CMD) [9]. Due to a lengthy period of the cooperation, the research activities can be divided in to three cycles: (1) the MIKE 11 re-engineering project, (2) the merging of MIKE 11 and MOUSE re-engineering project, and (3) the MIKE 1D project, as shown in sub-section 4.1-4.3 respectively.

4.1. The MIKE 11 re-engineering project

2 An engine is a basic part of simulation.

Participant observation. I studied the functionalities and code architecture of the MIKE 11 engine and its related product, i.e. the MOUSE engine. I also compared similarity of the source code between MIKE 11 and MOUSE engines. In the meantime, I conducted informal interviews with DHI staff members. I found that organisation of software development influenced the development of product line architecture [21].

Deliberating change. After finding a striking similarity in the source code between MIKE 11 and MOUSE engines, I presented a poster highlighting identical code parts in order to initiate a discussion among software developers. Subsequently, I made a presentation on software architecture and product line architecture making visible the benefits of merging MIKE 11 and MOUSE engines. Later, I participated in a subproject that discussed a new data access module architecture for the MIKE 11 re-engineering project and developed a prototype of the module.

Evaluation of the prototype and implementation efforts. When I organised an evaluation workshop with a group of DHI software specialists, we evaluated the flexibility of the new data access module. We also looked at different change scenarios at DHI and their implication in terms of implementation efforts. I continued my participant observations of the data access module subproject, the merging MIKE 11 and MOUSE engines re-engineering project, and the informal interviews with DHI staff members.

4.2. The merging of MIKE 11 and MOUSE engines re-engineering project

Participant observation. In order to be close to the project, I took on a task to create the architectural documentation. I had reviewed some of DHI's architectural documentation and online user references systems. I also observed development practices and technical infrastructure of the MIKE 11 and MOUSE engines. I reviewed a number of documentation generators which automate technical document production from the source code. I interviewed developers and internal users of MIKE 11 and MOUSE about how they can make use of the

architectural documentation. I found that architectural knowledge was more in the discussion than in the documentation.
Deliberating change. After analysing the code, I proposed a layered architecture to represent architectural knowledge. I reported a comparison of documentation generators and recommended a generator that was suitable for the project. I created a prototype of an online architectural knowledge system. The system contained a project overview, architectural knowledge, user references, and examples. The overview of the project explained the vision of MIKE 1D project. The architectural knowledge presented the overall design, layered architecture, and diagrams along with their precise explanations. The user references showed technical documentation, which was automatically generated from source code, e.g. class overviews, namespace overviews, and interface overviews. The examples described use scenarios of some components in a number of programming languages. The **evaluation** of this cycle was not possible because the merging project quickly moved to the MIKE 1D project. However, the prototype of the online architectural knowledge system has been set up and recently used internally.

4.3. The MIKE 1D project

Participant observation. I had reviewed static code analysis tools, e.g., [10], [13] and [5]. Employing those tools, I analysed the MIKE 1D source code and identified the relative complexity of the MIKE 1D components. I also compared the static analysis of the MIKE 1D project's source code with that of the MIKE 11 re-engineering project, and the merging of MIKE 11 and MOUSE engines re-engineering project. I continued informal interviews with MIKE 1D team members and joined the weekly meetings of the MIKE 1D project. I found that architectural analysis tools and technique embedded in daily routine were welcome by the development team.
Deliberating change. I and a group of software architecture experts conducted a workshop on architecture discovery with MIKE 1D team members. We also introduced the basic idea of checking for architectural conformity. We recommended tools for automated (1) checking source code and architecture at build time, (2) continuous

integration server [7], and (3) checking source code for proper format [15]. After finding the "good" and "bad" parts of the source code by using static code analysis tools, I presented my findings at the weekly meetings of the project. The source code comparative analysis was presented in the form of a Kiviat metrics graph [20]. The interdependencies between components were represented in a layered architecture and a dependency structure matrix [16].
Evaluation of the uses of architecture. I had interviewed MIKE 1D team members about how they can make use of the architecture as an aspect of software development. I continued with my participant observation of the MIKE 1D project. At the time of writing, the team members and I are planning a workshop on architecture-level evolvability assessment. Part of the workshop will present the online architectural knowledge system, which the first feedback on the system are expected to be given.

5. The concretisation of the architecture as the build hierarchy

This section explains on how MIKE 1D developers design MIKE 1D architecture and how the developers concretise the build hierarchy into the development environment, as shown in sub-section 5.1 and 5.2 respectively.

5.1. Designing architecture

Based on the architecture discovery workshop, the architecture of the MIKE 1D was explicitly defined. Figure 1 shows a sample of the MIKE 1D design architecture. Firstly, MIKE 1D developers divided the whole core computational part into a *Data Access* layer and an *Engine* layer. Secondly, the developers divided the *Engine* layer into two sub-layers: *Topology*[3] and *Pure Calculation*[4]. Thirdly, the *Data Access* layer is further divided into a number of components. So is the *Engine*. Later on, the developers focused on having the interfaces of the

[3] Topology handles a static model data, e.g. network topology.

[4] Pure calculation handles a dynamic model data which is used in the actual computations and simulation state.

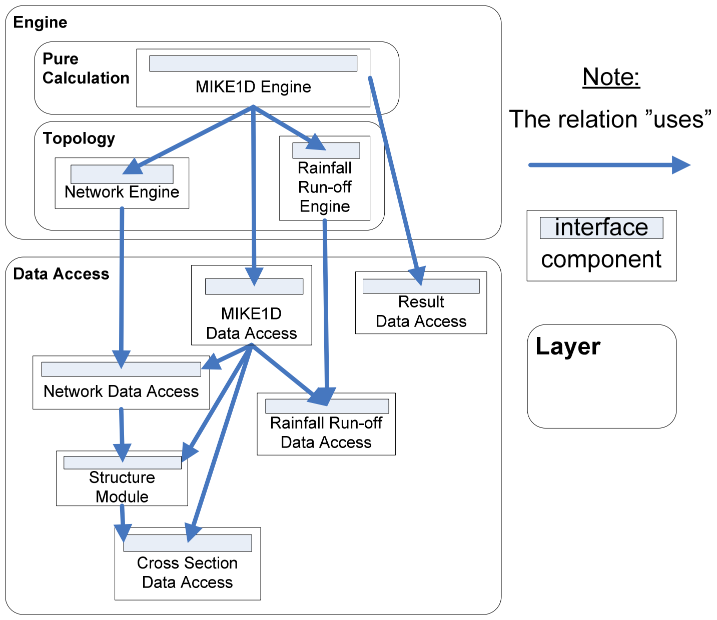


Figure 1. A sample of the MIKE1D design architecture

components as close to finished as possible. The interfaces identify how the components should communicate with each other. The developers logically categorised the components into the layers. Finally, the developers assembled everything based on design rules. The design rules defined acceptable dependency between components, i.e., (a) prohibition of upward relationships, it is inherent to layered architectures that references from lower to upper layers are not allowed – in other words, only downward relationships are allowed; (b) interface violations, usage of non interface artefacts of components by other components is not allowed; (c) several layer downward relationship is acceptable; and (d) prohibition of relationships within a layer, components in different line of products should not relate to each other. A build hierarchy is implied in the design architecture. The components in upper layers must be built after those in lower layers. For example, in Figure 1, the *Network Engine* component in the *Topology* layer must be built after the *Network Data Access* component, the *Structure Module* component, and the *Cross Section Data Access* component in the *Data Access* layer.

5.2. Concretisation of the build hierarchy into development environment

Part of the reluctance to work with an explicit architecture was the fear of having outdated documents and a diverging code base. Introducing the architectural compliance checking into the daily routine was thus welcomed by the development team. Currently, a build hierarchy is defined in such a way that developers specify the build order. But, Microsoft Visual Studio has another way of handling the logic of a build hierarchy. Microsoft Visual Studio has a *Solution*, a top collection of *Projects*. MIKE 1D developers work under the same *Solution*, i.e. the MIKE 1D *Solution*. In the *Solution*, there is a list of *Projects*. Each developer is responsible for his *Project(s)* in the MIKE 1D *Solution*. Each *Project* contains actual source code and its unit tests. Each *Project* represents a component. Hence, the developers define the dependency between components through the *Projects*. Afterwards, the developers can see in which order the *Projects* are built. When the developers compile or build the *Solution*, the build hierarchy will automatically check whether the developers have followed the design architecture. When the developers check out from the source control system and re-compile or re-build the *Solution*, they will be aware of what the other developers have been doing. The developers also use unit test to assure that any functionality change will not break the architecture.

Concretising the build hierarchy could be followed beyond Microsoft Visual Studio. A few examples of other build automation tools are GNU Make [11], Apache Ant [1], Apache Maven [2], and SCons [17].

6. Beneficent effects of build hierarchy

This section identifies advantages of concretisation of the architecture as the build hierarchy into software development. Confirmed by MIKE 1D team members, the advantages are categorised into (1) software quality and flexibility, (2) communication and cooperation to new developers, (3) distribution of work and parallel implementation, and (4) usage by hydraulic and environmental consultants, as shown in sub-section 6.1-6.4 respectively.

6.1. Software quality and flexibility

MIKE 1D developers iteratively work on the design architecture and keep on refining the architecture. For instance, with the help of the build hierarchy the developers are able to see if dependencies point to the wrong direction. In order to turn around the order of dependency, the developers will tweak a number of the other dependencies or introduce a new component. Consequently, the refined design architecture is reflected in the build hierarchy. Since the architecture has become modularised, each component can easily be tested thoroughly separately.

The core components can be replaced. For example, to change the equation of water flow in a core component, a developer implements a specialised component with the same interface as that core component. Without knowing how the core component is internally implemented, the developer only sees the interface of the core component and implements his specialised functionality. Afterwards, he moves the core components out, and replaces the core components with the specialised component without impacting anything else in the build hierarchy.

When the consultants needed a specific feature, the developers would look everywhere in the code and implement the specific feature even if the feature would benefit only that particular consultancy project. As a consequence, the software product will have changed between releases, and the specific feature is maintained, even though nobody else uses it. Also, the specific feature may give rise to additional work when the software product is upgraded or released in a new version. Even with interface in place, changes can degrade system if performance or other quality attribute of new version differs substantially from that of old version. With the design architecture, the developers can create a special component in a specific file for a specific feature and add it on independently of changes to other components. As long as the interface of the specific component does not change, the developers do not need to update the specific component.

6.2. Communication and cooperation to new developers

When new team members are introduced to the MIKE 1D project, their tasks are explained from an architectural point of view. The main developer shows the MIKE 1D design architecture, albeit not in much detail, but it helps the new members to start on the project. The new members can easily picture how the components will fit together. It is a strong point of the MIKE 1D project, one of the members said; the project has a "walking architecture[5]" that can go out and tell people about the architecture.

The new team members are initially assigned to implement a self-contained component[6] of the design architecture. Thus, the new team members will not change anything in the core components. The new developers have templates to start implementing. When asked how to figure out into which component he should put the physical equation, one of the developers who had been introduced to the team replied: "*Actually, the main developer showed me the component and told to put the equation here and here. Then I started it. For the framework of MIKE 1D, I didn't really know how it works. ...*". Another new member had experienced being a new developer in another project. He had to understand what the project was doing strictly by deciphering the source code. "*That was a time consuming task, but much less now after I moved to MIKE 1D…* ", he reported.

6.3. Distribution of work and parallel implementation

The idea of a build hierarchy is straightforward to the developers. MIKE 1D developers distribute the work after the architecture has been

[5] a.k.a. a chief architect or a main developer who carried most if not all the architectural knowledge and makes design decisions.

[6] A self-contained component refers to an independent component or a component that barely uses or is used by another component.

designed. They decide on a protocol for communication and dependencies between components up front. They can work on their own implementation without compromising each other's work. Before the concept of design architecture was introduced, functionalities were often mixed together in the same file and on the same unit. The developers would practically always write on the same files at the same time. Then, they would merge their changes. They would sometimes experience that merging a file was not an agreeable approach to all. After the introduction of architecture as a way of thinking, the process of using architecture more explicitly has decreased the number of conflicts. For example, the developers hardly ever work on the same file. Even though their source control system becomes better, they do not need to merge a file as often as before because they are working in separate components. One of the MIKE1D developers said, "*It's just ways easier to handle it. It is much easier to test. It is just a lot easier for us to work with, and it works better.*"

The idea of out sourcing MIKE 1D development to the other developers that know both MOUSE and MIKE 11 engines was mentioned. The developers can implement some of the components if they have time. They can work on it in parallel without affecting the rest of the MIKE 1D developers. A milestone and release plan has been decided from a functional point of view. However, thinking in terms architecture and build hierarchy has implicitly impacted on the plan.

6.4. Usage by hydraulic and environmental consultants

At the time of writing, the MIKE 1D project is still in the production phases, i.e. it is not yet finished; the developers are already beginning to see the benefits that MIKE 1D will eventually yield. With the flexibility in design, as mentioned in sub-section 6.1, the users, i.e. the hydraulic and environmental consultants will be able to replace any components without impacting on the whole software product. The users can tailor the core components by adding a specific component without changing any of the core components. Other than that, the users can change the non interface code of the specific component without

(a) waiting for the next release or (b) impacting the general software product.

7. Challenges towards evolvability

During the interview on perspectives of using architecture in development, MIKE 1D team members raised many interesting challenges towards evolvability. The challenges can be seen from different contexts, i.e., use context, technical infrastructure, technical selection, software engineering organisation, and software engineering practice.

Use context. The developers are in favour of generic programming. They design the MIKE 1D architecture in such a way that generic components are clearly separated from the specific components. Then, users, i.e. hydraulic and environmental consultants, can replace components without consequence problem. On the other hand, the users are in favour of "generic modelling". Separating the components does not guarantee that the users can model in the generic way. If the developers are not aware of the use side, it will be more annoying for the users. For example, there was a discussion about how to specify hydraulic resistance. Previously, the users had to specify the hydraulic resistance in many different files. Then, the first architectural recommendation was to do it in a specific way that required a lot of editing. But, the users would make more decisions and take time to set up their initial models. One of the users exaggeratedly said, "*In the way I understand it, the data should be located together with --- data and we could get it from that. You can do everything with it, but it will be terrible to work with the model in that way. You have repeated the number over and over again. That is not really the way to do it. ...I do not want to copy my information. I want to have it at one place that can be pointed in. Oh, it should be thirteen, not ten! I do not want to change 2,000 places. I want to change at one place. ...*". The truth of this matter is that the users have slightly more work in some respect, but the architecture reduces the users' work in another respect.

Technical infrastructure. MIKE 1D components are implemented in the C# programming language. But the software products predating MIKE 1D were implemented in the C++, C#, and Delphi programming languages. Handling the mixture of different languages would not be an easy task.

Technical selection. The ancestors of MIKE 1D have different approaches of implementing a similar functionality. In fact, the MIKE 1D's initial ambition is to make a common component between MIKE 11 and MOUSE. But there are some places where two approaches are equally good. For example, the time step calculation between MIKE 11 and MOUSE engines are different. MOUSE uses a smaller time interval, so the calculation takes longer time than in MIKE 11. Due to the general difference in length and time scales used by applications of MOUSE and MIKE 11 engines, they are optimised differently with respect to stability and accuracy. It is two different focuses that should be maintained. Although the developers try to merge the two approaches of implementing the same functionality as much as possible, they sometimes end up in the situation that "we need both".

Software engineering organisation. The software product department develops general purpose software products which can be sold to customers in great numbers. The consultancy departments have implemented very specific functionalities based on the needs of their projects. It is usually not possible to include a specific functionality into the generic products because (a) the functionality is not useful for the common user and (b) the functionality is implemented in a specific way rather than in a general sense. Thinking in terms of architecture is extremely important in the current DHI organisational structure. Without being aware of how the architecture looks, development in the consultancy department will diverge from that of the software product department. If the consultancy departments do not comply with the architecture in the software product department, they will potentially increase the maintenance as they may not be able to re-use components between releases.

Software engineering practice. Everyone in the MIKE 1D project can program and gradually learn about the architecture. But it always ends up that only a "walking architecture" can set up the architecture for the others.

One of the mentioned benefits is the interface-based design. But it becomes one of the weaknesses. Many interfaces have to be maintained. If the developers change the interfaces all the time, it will be difficult for the other people to work on any components. Therefore, the challenge is to define a viable interface of the core components as early as possible.

8. Discussion

This section notes general observations and findings, as shown in sub-section 8.1 and 8.2 respectively.

8.1 General observations

Relation between design architecture and code architecture. During the whole software lifecycle, architecture is being developed. Planned solution at the start of project usually signals the design architecture to control the code architecture. When the project is progressing, the code architecture might not conform to the design architecture with a good reason. For instance, the code architecture may reveal an infeasibility of the planed solution. Thus, the design architecture must be adjusted in order to align itself with the code architecture. A build hierarchy facilitates the code architecture conformance checking. The build hierarchy instantly reveals a divergent coincidence between the design architecture and the code architecture at the build time.
Waterfall process, iterative/incremental process, and agile methods. Development cycles – regardless of waterfall process, iterative/incremental process, or agile methods – are composed of analysis, design, implementation, and test. But time to complete a development cycle is what varied [4]. Moreover, a focus on the architecture in the design phase in each process/method is differed. The waterfall process requires the design of the complete architecture before the implementation, also called a “big design upfront”. The iterative/incremental process comes into the architecture at each design phase. The agile methods replace the “big design upfront” with “just enough design upfront” and “design as you go”. However, an architectural style [18] is taken as foundation of the agile methods.

8.2 Findings

Limitations of the build hierarchy. When I interviewed the MIKE 1D developers about the uses of the architecture, we came across the informative issues of the build hierarchy. The main developer put emphasis on keeping the architecture simple, neat, and explainable so that the developer can easily introduce a new developer to the architecture within a day or two. Having a strictly layered architecture on the build system is one way of doing it. Developers have a "preconceived" idea about the layers into which the components belong. However, the boundaries of layered architecture are rather implicit. If a new developer comes from outside and sees the MIKE1D's code architecture, he or she will not see the layer at first glance.

When the architecture has isolated functionality in several components, the connectivity between those components becomes complicated. The developers will have difficulty understanding how the functionality is invoked. The build hierarchy gives an overview of how the components are connected and dependant on each other. But it does not tell "what and when" the components are invoked unlike the UML diagrams, more specifically, class and sequence diagrams. The build hierarchy initially follows the design architecture, as noted in architecture documentation, but both of them are easily out of sync during development.

Acknowledged eXtreme programming practices and component-based software engineering. The eXtreme programming (XP) is one of the most mature and best-known agile practices [14]. Some of the practices have been used in the MIKE 1D project, e.g. a unit testing and an open work space. The unit testing and the build hierarchy play a mutual role in order to facilitate the code architecture conformance checking. When the developers change something in the component that will effect or fail somebody else's component, the unit test and the build hierarchy will capture that effect and instantly notify the developers.

Confirmed by the MIKE 1D team members, the open work space is one of the important elements that promotes decentralisation of the architectural knowledge during the development. The members work in a common "airy" room where they can sit near each others. When it comes to architectural discussion, the "roommates" easily perceive "what and why" the architecture has been changed.

A considerable extent of component-based software engineering provides flexibility for handling changes at the level of design architecture. A component interface design helps the MIKE 1D developers introduce new functionality with the minimum impact with regard to all changes. However, it is important that a "walking architecture" keeps an eye on any changes that could break the interface.

Increasing awareness of architecture in the evolution of software product line. Handling the evolution of software assets in a product line is more intricate than that of a tailored product. Supported by the DHI case study and [6], the main reasons are (1) most of the assets rely upon various software products and versions, and (2) multiple organisational units are involved. To maintain an overview of the status of the asset base, the build hierarchy significantly increases awareness of architecture. When the software products are upgraded or released in new versions, changes at the asset base must comply with the previous release versions. With the help of interface-based design, if developers want to change some component in the assets base, the developers create a new component with the same interface as the previous version. When the developers build software with the new component, the build hierarchy will notify developer whether the new component complies with the architecture of the previous releases. Thus, maintainability, one of MIKE1D quality attributes, is influenced positively by the increased separation of components. However, the management among multiple organisation units should be optimised in parallel.

9. Conclusion and requirements for the architectural analysis tools

Architecture is taken as a foundation of many development processes or methods, but the use of explicit architecture throughout the software life cycle is hardly ever taken seriously, especially in the agile methods. But, this work shows that after the architecture was concretised as the build hierarchy, the architecture becomes the "first class citizen" of the software development. Whenever developers build the software, the development environment immediately notifies the developers of the compliance between the design architecture and the code architecture. However, the build hierarchy does not give any information on “what, when and why” the components are invoked. Confirmed by DHI case study, the concretisation of the architecture as the build hierarchy has improved, as shown in Section 6, (1) the software quality and flexibility, (2) the communication and cooperation to new team members, (3) the distribution of development tasks and parallel implementation, and (4) the foreseen usage by hydraulic and environmental consultants.

Based on the case study, we reveal requirements for the architectural analysis tools that are meant to be embedded in the daily development practice. The tool should (a) give frequent feedback, the compliance between the design architecture and the code architecture should notify the developers at the build time; (b) not show only dependencies, each line of the dependency or the “uses” relationship should be displayed further in the UML diagrams; (c) support different programming languages, the software products often invoke or build on top of legacy objects; and especially (d) support product line architecture, relationships among multiple software products and releases are complex – any changes at the core components effect not only different products but also different releases. The tool arising from the build hierarchy could pragmatically promote the application and awareness of architecture throughout the whole software life cycle.

References

[1] Apache Ant website, http://ant.apache.org, last visited 10-02-09.
[2] Apache Maven Project website, http://maven.apache.org, last visited 10-02-09.
[3] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
[4] K. Beck, "Embracing Change with Extreme Programming", *IEEE Computer*, October 1999, pp. 70-77.
[5] W. Bischofberger, J. Kühl and S. Löffler, "Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking", *Proc. 1st European Workshop on Software Architecture (EWSA2004)*, *LNCS 3047/2004*, Springer-Verlag, Germany, 2004, pp. 1-9.
[6] Bosch, J., *Design and Use of Software Architectures: Adopting and evolving a product-line approach*, Addison-Wesley Professional, 2000.
[7] CruiseControl.NET website, http://sourceforge.net/projects/ccnet, last visited 24-01-2009.
[8] DHI Water Environment Health website, http://www.dhigroup.com, last visited 21-01-2009.
[9] Y. Dittrich, K. Rönkkö, J. Erikson, C. Hansson and O. Lindeberg, "Co-Operative Method Development: Combining qualitative empirical research with method, technical and process improvement", *Empirical Software Engineering Journal 13(3)*, Kluwer Academic Publishers, 2008, pp. 231-260.
[10] Lattix website, http://www.lattix.com, last visited 21-01-2009.
[11] GNU Make website, http://www.gnu.org/software/make, last visited 10-02-09.
[12] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation", *IEEE Transactions on Software Engineering 27(4)*, IEEE Computer Society, 2001, pp. 364-380.
[13] NDepend website, http://www.ndepend.com, last visited 21-01-2009.
[14] R.L. Nord, and J.E. Tomayko, "Software Architecture-Centric Methods ad Agile Develoment", *IEEE Software 23(2)*, IEEE Computer Society, 2006, pp. 47-53.
[15] NStyle website, http://www.vadesoft.com/help/help.htm, last visited 24-01-2009.

[16] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture", *Proc. 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications*, ACM, 2005, pp. 167-176.
[17] SCons website, http://www.scons.org, last visited 10-02-09.
[18] M. Shaw, and P. Clements, "Toward boxology: Preliminary classification of architectural styles", *Proc. 2nd Int. Software Architecture Workshop*, ACM, 1996, pp. 50-54.
[19] D. Soni, R.L. Nord, C. Hofmeister, "Software Architecture in Industrial Application", *Proc. 17th Int. Conf. on Software Engineering (ICSE'95)*, ACM, 1995, pp. 196-207.
[20] SourceMonitor, Campwood Software website, http://www.campwoodsw.com, last visited 22-01-2009.
[21] H. Unphon, and Y. Dittrich, "Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture", *Proc. IASTED Int. Conf. on Software Engineering*, Innsbruck, Austria, Feb. 2008 , pp. 178–183.

SOFTWARE ARCHITECTURE AWARENESS IN SOFTWARE PRODUCT EVOLUTION

Hataichanok Unphon and Yvonne Dittrich

IT University of Copenhagen,

Rued Langgards Vej 7, DK-2300, Copenhagen S, Denmark
{unphon, ydi}@itu.dk

ABSTRACT. Software architecture has been established in software engineering for almost 40 years. When developing and evolving software products, architecture is expected to be even more relevant compared to contract development. However, the research results seem not to have influenced the development practice around software products very much. The architecture often only exists implicitly in discussions that accompany the development. Nonetheless many of the software products have been used for over 10, or even 20 years. How do development teams manage to accommodate changing needs and at the same time maintain the quality of the product? In order to answer this question, semi-structured interviews were conducted in order to find out about the wide spectrum of architecture practices in software product developing organisations. Our results indicate that a chief architect or central developer acts as a 'walking architecture' devising changes and discussing local designs while at the same time updating his own knowledge about problematic aspects that need to be addressed. Architecture documentation and representations might not be used, especially if they replace the feedback from ongoing developments into the 'architecturing' practices. Referring to results from Computer Supported

Cooperative Work, we discuss how explicating the existing architecture needs to be complemented by social protocols to support the communication and knowledge sharing processes of the 'walking architecture'.

KEYWORDS

Cooperative and human aspects, software architecture, architecture knowledge management, qualitative empirical studies

1. Introduction

Software products are programs that are used by more than one organisation. They are often configured and customised to fit with the specific use context. They are long-living; often evolving over several decades. Bug fixes and upgrades are delivered on a regular basis. The study presented here has been motivated by research with several product developing companies [25, 26, 70]. Though especially for software products that need to evolve to keep up with technical and application domain developments the architecture should be an important asset, our previous research showed that the companies we have been in contact with did not have a formal architecture process. Nonetheless, the software was successful over long periods of use and evolution. So our questions were: how do these companies manage to maintain the evolvability of their products over a long life-time?; Are the development teams aware of the architecture of their product? If yes, how does the architecture knowledge become influential in the development, and how is the architecture updated in the ongoing evolution?

The notion of awareness was first used by Heath and Luff in their analysis of the cooperation of line controllers and Divisional Information Assistants in line control rooms of the London underground [29]. Through both monitoring common displays and each other's activity, they managed the common tasks–advising train drivers and informing passengers about delays–with little explicit coordination. Even the reactions of each colleague was monitored so that the missing of a necessary reaction could be caught, which then caused either a more emphasised behaviour or if necessary explicit coordination. Since then, such heedful situated coordination has been reported from a range of activities and resulted in specific awareness support in groupware applications [24]. Schmidt remarks on the topic in an article for a special issue of CSCW journal: awareness is a substantiation of an attribute of an activity [60]. Someone is acting in awareness of the activity of others and of changes in the context. The issue then is to understand what the coordination or awareness mechanisms in play are, and how they can be supported. In other words, what are the clues a co-operator is reading, how does he or she make their own actions accountable to the surroundings, and what are the means and protocols in play?

Applying the concept of awareness on software architecture indicates a focus on the daily use of architectural knowledge to a.) make changes to a module that might have implications on another's code – that is changes the interface – visible, b.) monitor changes that are relevant for the task at hand, and c.) monitor changes to the code, the requirements, and the context that makes it necessary to change the architecture and thus change the design and implementation of the different modules.

The article reports results of an interview study. Though awareness mechanisms in action need to be observed in practice, interviews provide the possibility to compare the practices reported about different companies, and thus give an indication of whether or not we are looking at a wider spread phenomenon. We interviewed members of eight software developing organizations in five countries (i.e., Belgium, China, Denmark, Germany, and Switzerland). Each organisation has ongoing software product development. The interviews were done and analysed in a grounded theory manner. The motivations of applying a grounded theory approach was to not only collect information of what is going on in the companies, but also what motivates different practices, and how they depend on each other.

Our results indicate that the industrial practice in most cases is not what is recommended by applicable textbooks. Nonetheless, the structure of software products is regarded as an important asset of development. Rather than documenting it in a formal way, most companies rely on what we've begun to call a 'walking architecture.' This is a key person, or a number of key persons, who maintain and update the structure of the software, and are involved in discussions of changes motivated in the development, or by new requirements, and who introduce newcomers to the structure of the software. Representations of the architecture thus are temporary and partial, e.g., sketches on whiteboard and scrap paper used in a specific situation. The result of this practice is not only the distribution of architectural knowledge to the development team, but also an update of the chief architect's knowledge on the issues the developers meet when they develop.

We argue in our discussion that software architecture literature, so far, has underestimated this feedback, and that here maybe we can find reasons for the lack of appreciation of recommended software architecture methods in industry. By using the awareness concept from CSCW when discussing our findings, we highlight the importance to focus not only on documentation and tools when improving architectural practices, but also on the development of social protocols around such methods and tools.

In the next section, we discuss related literature on software architecture and software evolution, but also on knowledge management in software engineering and the concept of awareness that originated in the discourse of computer supported cooperative work that has been adopted in research on distributed software engineering. Section 3 shows research methodology. Section 4 elaborates on interviewees and their companies, then briefly describes interview guideline. Section 5 shows an analysis of software architecture awareness. Section 6 is discussion. Section 7 is conclusions.

2. Architecture, knowledge, and awareness

This section introduces the research we build upon and contribute to. The section starts with discussing the notion of software product architecture and evolution, then discusses knowledge management in software engineering and the notion of awareness which stems from the discourse on computer supported cooperative work.

2.1. Software architecture

In programming, the term architecture has been used since the late 1960's [7]. In the early 1970s, Parnas contributed many of the fundamental tenets and principles behind software architecture [52-55]. The report and book by Garlan and Shaw [18, 62] not only redefined software architecture overall, but also introduced a number of architectural styles and reference architectures. They introduced the notions of components, connectors and constraints, emphasised the notion of the design rational, and illustrated how architectural representations can improve the understanding to complex software systems.

Software architecture is meant to serve a number of purposes: as it decomposes the software into components, it helps to handle complexity in a divide and conquer manner; the decomposition serves also as a base to structure the implementation work into manageable chunks assigned to individuals or small teams; it provides a base to analyse and assess non-functional requirements of the software to be built, or of the changes introduced [3, 33, 64].

The practices proposed heavily rely on written representations, and when necessary, (semi-) formal notations, e.g., [1, 2, 5, 14, 15, 19, 20, 41, 42, 44, 63, 67, 74]. The notations provide an explicit way of specifying the elements and their connections used in the architecture[1]. Over time, as the software evolves, the code structures become less tightly coupled with the design architecture (a.k.a. the code view vs. the module view [32]). The design architecture has layers, modules and dependencies, but the source code architecture contains folders and files, as well as, static and dynamic relationships between different classes. Keeping the correspondence between design architecture and code architecture alive requires a rigorous engineering discipline.

2.2. The role of the software architect

Literature places the main responsibility for the maintenance of the architectural structure of a software product with the software architect. The role of the software architect has been discussed mainly based on substantial experience. The roles and responsibilities of an architect or

[1] Tools developed on top of those notations generate part of the implementation based on the architecture. In this way, the organisation of source code—when developed that way from scratch—conforms to major design elements and the relationships among them. Model Driven Development (MDD) aims at supporting the evolution through these tools [9, 23].

an architecture team listed in [34] are (*i*) defining the architecture of the system; (*ii*) maintaining the architectural integrity of the system; (*iii*) assessing technical risks; (*iv*) working out risk migration strategies/approaches; (*v*) participating in project planning; (*vi*) proposing order and content of iterations; (*vii*) consulting with design, implementation, and integration teams; and (*viii*) assisting product marketing and future product definitions. The definition of software architecture includes all the usual technical activities associated with design: Understanding requirements and qualities; extracting architecturally significant requirements; making choices; synthesizing a solution; exploring alternatives and validating them; etc. For certain challenging prototyping activities, architects may have to use services of software developers and testers. The maintenance of the architectural integrity takes place through regular reviews; writing guidelines, etc. and presenting the architecture to various parties as different levels of abstraction and technical depth. For many effort estimation aspects, or for the planning of distributed development, managers need the assistance of architects. Because of their technical expertise, architects are drawn into problem-solving and fire-fighting activities that are beyond solving strictly architectural issues. The architects have insights into what is feasible, doable, or 'science fiction' and their presence in a product definition or marketing team may be very effective. However, good architects should bring a good mix of domain knowledge, software development enterprise, and communication skills.

Fowler [17] categorised architect's roles into two types: *Architectus Reloadus* and *Architectus Oryzus*. *Architectus Reloadus* is an architect who makes all the important decisions. The architect in this type does this because a single mind is needed to ensure a system's conceptual integrity, and perhaps because the architect doesn't think that the team members are sufficiently skilled to make those decisions. Often, such decisions must be made early on so that everyone else has a plan to follow. *Architectus Oryzus* is an architect that must be very aware of what is going on in a project, looking out for important issues and tackling them before they become a serious problem. The most noticeable part of the work for *Architectus Oryzus* is the intense collaboration. By way of illustration, in the morning, the architect programs with a developer, trying to harvest some common locking code. In the afternoon, the architect participates in a requirements session, helping explain the technical consequences of ideas, such as development costs, in non-technical terms. The most important activity of *Architectus Oryzus* is mentoring the development team to raise their level so that they can take on more complex issues. Improving the development team's ability allows an architect much greater leverage utilizing the entire team rather than being the sole decision maker and running the risk of becoming an architectural bottleneck. This leads to the rule of thumb that an architect's value is inversely proportional to the number of decisions he or she makes.

Based on more than ten years of experience, Kruchten has recommended a simple time-management practice for architects [35]. The recommended time ratio allocates 50% *internal*, 25% *inward*, and 25% *outward* activities. The *internal* activities focus on architecting per se (architectural design, prototyping, evaluating, documenting, etc). The *inward* and *outward* refer to cooperation and communication with other stakeholders that the architects interact with. The *inward* is to get input from the outside world. For example, listening to customers, users, product manager, and other stakeholders (developers, distributors, customer support, etc.), and learning about technologies, other system's architecture, and architectural practices. The *outward* can be seen as providing information or helping other stakeholders or organisations (e.g., communicating architecture, project management, or product definition). The 50:25:25 time-management ratio helps the architects to be aware of the risks of falling into one of the following situations: creating a perfect architecture for the wrong system, creating a perfect architecture that's too hard to implement, architects in their ivory tower, or absent architects.

2.3. Software product evolution and architecture

The intrinsic evolutionary nature of real-world computer usage and of software embedded in its use context was originally recognised in [4, 37]. Lehman defines *E*-type systems as operating in supporting an activity of the real world. The dynamism of the real world induces software to be continually changed, updated, and evolved over its life-time. As the software evolves, its architectural integrity tends to dilute. For example, source code architecture drifts from its design architecture [46]. The gap between the source code and the design architecture hinders program understanding which leads to development and maintenance activities that are increasingly difficult and highly error prone [68, 69]. However, the effort emphasised on updating the software in order to improve upon the future maintainability without changing its current functionality, the so-called 'preventive maintenance' [39] (a.k.a. anti-regressive activity [38], or refactoring [16]), is not highly prioritized [40, 59]. Even though preventive maintenance offers significant improvements in the simplicity of conducting maintenance interventions in the long term, it brings little to no immediate benefits [43]. As a result the software becomes more difficult to maintain.

Decisions made during initial software development affect the ability of organisations to successfully perform software change. In particular, the selection of architecture could either aid or hinder changes made through evolution [43]. In software product line engineering, most case examples [6, 8, 73] report on the strong role of its architecture. The case examples rely on companies developing product lines of technically embedded systems. The product line designs are constrained by hardware and mechanical parts whose specification and variability are known in advance. Product line architecture is a common architecture for a set of related products or systems developed by an organisation. Designing such an architecture supports variability by taking the diversity of the technical environment into account. Once the first version of the product line architecture, along with the sets of components and products have been developed, the evolution of these assets will become the primary activity.

Most of the companies we interviewed develop socially-embedded software[2]. The variability needed to accommodate is provided by configuration mechanisms [57]. However, this variability does not completely resolve the need of evolution, as it only implements support for anticipatable variability. To react on un-anticipated, evolving needs, the software itself evolves over time.

2.4. Knowledge management

Knowledge management has been a major topic in software engineering, even before the concept had been coined by Nonaka at the beginning of the nineties [49, 50]. In their seminal article 'The rational

[2] Socially-embedded software refers to software that can be modelled intensively according to the environment and practices of its end-users [72].

design process: Why and how to fake it', Parnas and Clement argue that though to design software by deriving the design and source code from the requirements is not possible, the software team should aim at producing the documentation that mirrors such a rational design process [56]. The reason is to document the reasoning and rationale behind design decisions in case revisions become necessary, and in order to have suitable documentation for the maintenance team. Today this argument could be related to what Dingsøyr and Conradi call codification approaches to knowledge management [13]: such approaches emphasis the codification, digital storage, and retrieval of information representing relevant knowledge. Complementary personalisation oriented approaches emphasize face-to-face communication between people in-the-know and who need to know. In his article 'Programming and Theory Building', Peter Naur [48] emphasises the importance of participation in the development team to understand the rationale behind a design. Only through participation in design and development can help software engineers to understand how the software models its problem domain and supports the needs of its users.

Though software engineering from the beginning emphasised the importance of documentation for the development process, and therefore exhibited an affinity to codification oriented approaches to software engineering, empirical research indicates that face-to-face knowledge sharing is at least as important. In a dialogue situation, knowledge can be tailored to the context in which it is needed. Through this communication, the software engineer who shares his knowledge, also updates his knowledge.

The discussion on software architecture knowledge emphasises the codification, storing and retrieval of information on architecture. However, this codification strategy does not work in practice [36]. The people involved in the architecting process (who own the knowledge) often do not document it [28]. The reasons are a lack the motivation to document and maintain architecture knowledge, as the benefits do not seem substantial enough to justify the effort; the short-term interest in the project becomes more important than the long-term architectural knowledge reuse; developers are absorbed in the creative flow of design and thus don't reflect on long term impact of decisions; lack of training. Even worse, when the architecture knowledge is documented, it's often not sufficiently shared within the organisation. As examples

[28] gives (i) the knowledge is not disseminated to the appropriate stakeholders; (ii) the recipients of knowledge don't use it in their own tasks, either intentionally, or because there is no provision in the processes; (iii) it's cumbersome to search and locate the appropriate knowledge and adapt it in one's needs.

2.5. Awareness in software engineering

The concept of awareness as developed in the introduction highlights what can be called a situated socialisation-based knowledge sharing mechanism. In software engineering, the notion of awareness is so far mostly used to address coordination of distributed development. Storey et al. [66] give an overview of different tools that are designed to help programmers monitor changes to the common software under development that might become relevant for their own programming. Particularly, in spatially distributed development, parallel ongoing work cannot be monitored by means of 'overhearing' design discussions taking place in the vicinities. Also, meetings, such as daily stand-up meetings, that are designed to provide a project team with a development overview, with respect to the common product, cannot help this need. Tools visualising social-technical dependencies, and thus helping to contact the correct person (e.g. [11]), are designed to address this lack. Such tools can be understood as support for fine-grain knowledge sharing. Recent research, however, indicates that technical support addresses only one side of the problem: awareness problems can also occur in cases of mismatched social protocols [10].

The importance of such protocols has already been indicated in the early studies on the London underground control room. 'However, it is clear that whilst certain activities are primarily accomplished by specific categories of individuals, the *in situ* accomplishment of these tasks is sensitive to, and coordinated with, the actions and responsibilities of colleagues within the immediate environment. The competent production of a range of specialised individual tasks within the Control room is thoroughly embedded in, and inseparable from, a range of socio-interactional demands' [29, p. 82, highlighting by the authors]. This interlacing of their own and their co-workers activities is not only guided by a codex of explicit rules, but depends on competence with respect to understanding established practices [29, p. 78].

The usefulness of the technology that is the base of this interaction 'relies upon a collection of tacit practices and procedures through which Controller and Divisional Information Assistant (DIA) coordinate information flow and monitor each other's conduct [29, p. 87, see also 60].

With the term 'practice,' we describe a common way of acting acknowledged by the community that shares the practice [27, p. 1296]. A group of co-operators maintains the common practice through reproducing it in their every day actions. Practice thus is distinguished from ad-hoc behaviour, which as such is only perceivable by its deviation from both the formalized rules and the established practice. Such practices as well as explicitly agreed on procedures have been also called social protocols [21, 61]. They are developed and maintained through ongoing 'articulation work' or 'meta-work' of the members and can only to some extent be designed from the outside.

So far, only one project addresses awareness issues with respect to software architecture: application programmer interfaces (APIs), the interfaces that make the functionality of one module available, are discussed in a way that can be regarded as implementing the material side of an awareness mechanism [12]. Often when APIs are used to indicate boundaries between development groups, corresponding social protocols are established indicating that software teams who implement functionality using the module are informed if the API needs to change. The article, however, does not refer to the role of the architecture, nor the everyday work of the software architect.

3. Research methodology

The study has been designed as triangulation for in depth ethnographically informed studies in two organisations. Being aware of those architectural practices in situ would be best observed by participatory observation, we therefore decided for an interview study. The interviews [58] aimed at mapping out the architectural practices as well as documents and artefacts and their usage. The questions cover business contexts of the software products, development process, architecture, dimension and use of the architecture, cooperation of development, and awareness of change in the software, software product line, and software evolution. Our interviews focused on understanding concrete practices rather than a polished record, as we focused on both the tools used and concrete occasions. We transcribed the interviews and analysed them in a grounded theory manner. This section is outlined as follows: Sub-section 3.1 presents grounded theory; Sub-section 3.2 presents interviews as data collection; and Sub-section 3.3 shows analytic process.

3.1. Grounded theory

Among flexible research strategies, grounded theory claims that it is useful in new, applied areas where there is a lack of theory and concepts to describe and explain what is going on. Grounded theory approach was derived from a combination of Chicago style Interactionism and Pragmatism [22], in terms of data collection and analysis. Data analysis focuses upon concepts. It is achieved by carrying out three kinds of coding: open coding to find concepts; axial coding to interconnect them; and selective coding to establish core concept(s). The result of this process of data collection and analysis is a substantive theory relevant to a specific problem, issue, or group [58, see also 17].

In grounded theory, data analysis is an iterative process. Open coding results in a set of initial concepts about the phenomenon. For each concept, sub-concepts referring to properties and dimensions are established. For axial coding, the data is assembled in relation to the concepts developed in the open coding. Selective coding central phenomenon, or a core concept, explores causal condition, or concepts of conditions that influence the phenomenon. In the selective coding phase, researchers integrate the concepts of the axial coding in a model that will reveal conditional propositions or hypotheses in the selective coding phase. Data collection and analysis continues until no new concepts and relations are found in the new data.

According to Robson [58], typical features of grounded theory are (*i*) applicable to wide variety of phenomena; (*ii*) commonly interview-based; and (*iii*) a systematic, but flexible research strategy which provides detailed prescriptions for data analysis and theory generation.

The problems in using grounded theory are (*i*) that it is not possible to start a research study without some pre-existing theoretical ideas and assumptions; (*ii*) there are tensions between the evolving and inductive style of a flexible study and the systematic approach of grounded theory; (*iii*) it may be difficult in practice to decide when categories are 'saturated,' or when the theory is sufficiently developed; and (*iv*) grounded theory has particular types of prescribed categories as components of the theory which may not appear appropriate for a particular study.

3.2. Interviews

For the interviews, we contacted eight software product development organisations in five countries, i.e. Belgium, China, Denmark, Germany and Switzerland. Each organisation had its own ongoing software product development. Sizes of the interviewed organisations ranged from a three-person organisation, to an international organisation with more than 20,000 employees. The sample included normal industrial product development as well as an open source project lead by a governmental agency, a research institution, and a semi-private research and consultancy company. Interviewees included a managing director, a chief technical officer, a chief architect, a senior consultant, a group leader, and a number of software developers. Most of the interviewees did not want to disclose their name and organisation name, thus, we present them under assumed names.

We prepared an interview guideline with two parts, i.e. a number of open questions, and a series of multiple-choice questions. The free response questions addressed the six categories listed in the introduction of this section. The fixed questions were asked after the open question. The answers thus quantified what had been discussed during the interview.

The interviews were conducted from late 2007 until early 2008. The duration of each interview varied between 30 minutes and three hours, depending on the interviewee. The interviews were audiotaped and transcribed. The transcription and analysis of the interviews were checked by the interviewees. Apart from that, we also provided confidentiality agreements for the interviewed organisations.

3.3. Analytic process

The analysis process started with reading each transcription in order to get a feeling of what the interviewees were telling. We came back and listened to the voice recordings of the interviews while reading the transcriptions. The first tentative concepts were presented as memos and a springboard of our analysis. These codes were then highlighted in the transcription together with words or terms that supported properties and dimensions of the concept. During this work, higher-level categorisations, or lower-level explanatory concepts became visible. With this list of concepts and codes, we proceeded to the next transcription. New concepts appearing in later interviews were added to the coding scheme which required returning to previous interviews. This part of the analysis process continued until we were satisfied that we had accounted for the contents of the interview.

Relations between different concepts became visible. In order to fully understand concepts, we explored the semantic and process context of the concepts in the interviews. For the semantic context, we explored conditions and relationships between the concepts the interviewees expressed in the interviews. For the process context, we explored how interviewees responded to concepts through action, interaction, and emotions. That way, we further explored the meaning of the concepts and linked the concepts to one another.

We represented the relationship between major categories and subcategories as the foundation for the theoretical structure that we iteratively refined by going back to transcripts and memos. Figure 1 shows such a representation of how the categories referred to each other. Section 5 presents the central concepts, and also indicates how they relate to each other. Section 4 provides the background of the organisations and the results of the closed questions asked at the end of the interview.

3.4. Credibility

The strategies to minimise possible threats to validity [58] and to enhance trustworthiness shows as follows:

Triangulation. The term triangulation, borrowed from navigational science and land surveying, referred to using two or more sources to

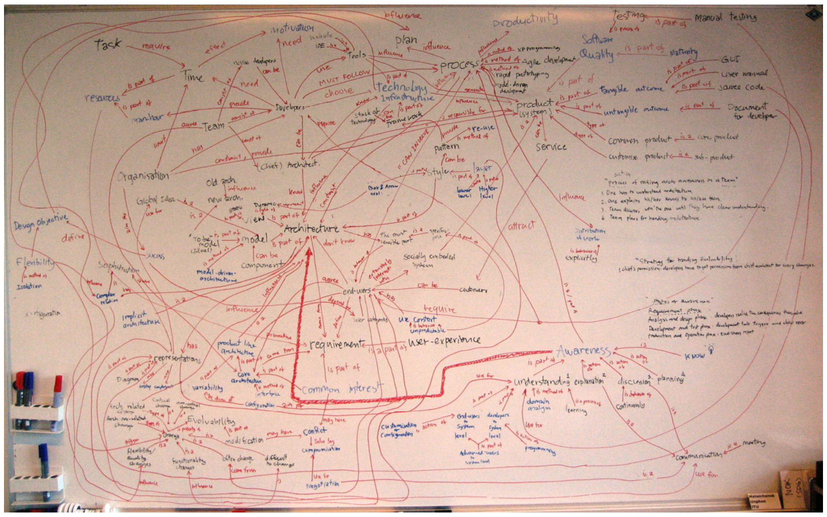


Figure 1. Early diagram over analysis concepts regarding architecture awareness

achieve a comprehensive picture of a fixed point of reference [51 , p.186]. The data were triangulated from 13 interviews (given by 15 interviewees from eight different product developing organisations). The interview study was stop when concepts, categories and theoretical structure were saturated.

Member checking. All interviewees were required to check the transcription of their interview before the data was analysed. This article was reviewed by the interviewees before submission.

Audit trail. All interviews were audio-taped and transcribed. During data collection and data analysis, drawing artefacts and diagrams on the whiteboard were photographed. Detailed analytic and self-reflective memos were documented.

4. Background

This section describes the base of the analysis. Sub-section 4.1 presents interviewees and organisation profiles. Sub-section 4.2 outlines dimensions, sophistications, and states of architecture.

4.1. Interviewees and organisation profiles

There are 15 interviewees from eight organisations developing and maintaining their own software products. Table 1 gives an overview over interviewees and companies. Note that the order of interviewees follows the chronological order of the interviews.

The first company is EW, a Belgian government agency in the Walloon region. The main task of EW is to simplify the lives of citizens and enterprises that need to communicate with public entities. EW develops and acquires 20 software products and projects for e-government, and on-line public administration. EW has a total of 22 employees, ten of whom are in IT department. EW employs the core contributors for an open source project following a product-line approach that is developed together with a number of IT people in different municipalities. We interviewed Gaëtan, a senior developer educated in computer science who has been working at EW for two years. He is the main developer of the open source project and cooperates with a Belgian university to explore Model Driven Development (MDD), along with feature modelling techniques. The project started with evolving a product family for advanced meeting

Table 1. Summary of sampled organisations

No.	Company name	Software product industry	Total employees	Interviewees
1.	EW	E-government applications	20-22	one senior software engineer
2.	ABC	Hydraulic simulation	800	five senior software engineers, and two offshore junior software engineers
3.	OMD	Business identity management	72	one junior software engineer and one senior consultant
4.	ARG	CRM and telecommunication	3	one managing director
5.	GDT	Computer security	51-200	one senior software engineer/chief architect
6.	CO	Visual office solutions	10	one chief technology officer
7.	XYZ	Internet searching and organising universal information	20,000	one software engineer
8.	DZ	Controlling cryogenic processes for colliders	1,001-5,000	one group leader and one software engineer

management functionality, like meeting workflow specifications and document generation.

The second organisation is ABC, an independent research and consultancy in the field of water, environment, and health. The company has approximately 800 employees, and is based in more than 25 countries worldwide with their headquarters in Denmark. ABC develops more than 15 commercial software products supporting water resources management, with the main expertise in hydrodynamic simulation. Some of the software products have been evolved for more than 20 years. Of the 35 developers, we interviewed five senior developers at their headquarters, and two offshore developers in China. One of the senior developers is the head of development and responsible for all products. The rest are responsible for different products, shown as ABC product#1-5 in Table 2. The five senior developers are educated in hydraulic engineering, while the two offshore developers are educated in IT. One of the interviewed developers is working closely with us on a project that re-engineers a core computational part of three existing products using a software product line approach. However, architectural tools and practices were introduced to the project after this interview study.

The third organisation is OMD, a Danish founded company that provides software and consultancy services for identity management, specifically delivering tools for Role Based Access Control (RBAC) and assisting organisations to comply with IT access regulations. Established in 1999, OMD has operations in Europe, Africa, Australia and North America, delivering its solution via a network of skilled partners and system integrators. The company has 72 employees: two in USA, three in Germany and the rest in Denmark. The IT team consists of 18 people, 12 of whom are developers. The company offers three standard software products of which only two are maintained. We interviewed two employees, to be hereafter named as Santiago and Neeraj. Santiago is a senior consultant, specialised in Business Process Management and educated in Economics. Neeraj is a junior software developer educated in IT.

The forth organisation is ARG, a private consultancy company in the field of customer relations management and telecommunications. For the past two years, ARG has been developing software products on top of call recording systems and CRM systems. We interviewed Ole, a managing director and a founder of ARG. ARG’s office and

Table 2. The presence of architecture with respect to software products

			Software products											
			EW Products	ABC Product #1	ABC Product #2	ABC Product #3	ABC Product #4	ABC Product #5	OMD Products	ARG Products	GDT Products	CO Products	XYZ Product	DZ Products
Different dimensions of architecture representation and usage	In which form is the architecture presented in the process?	In somebody's head	X		X		X	X			X	X	X	X
		Documented in folder, binder, or internet	X	X		X		X		X	X		X	
		Readily available in workspace		X				X	X		X			X
	How is the architecture represented?	Text	X		X	X		X		X	X			X
		Source code	X	X	X	X	X	X	X	X	X	X	X	X
		Boxes and arrows	X	X		X				X	X	X		
		Class, packages, and diagrams	X	X	X			X	X	X		X		
		Architecture Description Language (ADL)	X											
		Different views	X						X			X		
	How the architecture representation is used?	Design	X		X	X	X	X	X	X	X		X	X
		Communication between developers	X	X		X	X	X	X	X	X	X	X	X
		Communication about changes	X	X		X	X	X		X	X		X	X
		Communication when designing new features	X	X		X	X	X	X	X	X		X	X
		Communication for bug fixing	X			X			X	X			X	
		As feedback for ongoing implementation	X			X		X					X	X
		Distribution of work and responsibility	X		X	X	X	X		X	X	X	X	X
		Generation of diagrams	X					X	X	X				
	How is the architecture representation updated?	Never			X	X	X					X	X	X
		Regularly controlled	X	X				X	X	X	X		X	X
		Continuously		X					X		X			X
		Related to an overall plan and release plan	X						X					
		Only when problem occurs	X	X				X					X	
To what detail is the architecture represented?	Overall		X	X	X	X	X	X	X		X	X	X	X
	Classes		X	X		X	X	X	X	X		X	X	
	Styles		X	X								X		
	Patterns		X	X				X	X	X		X	X	
	Design patterns		X						X	X		X	X	X
	Various views		X	X									X	
How is the architecture expressed?	Implicitly	Source code	X	X	X		X	X			X	X	X	X
	Explicitly	Diagram	X	X	X			X	X		X	X	X	
		Textual description	X								X			
	Requiring substantial knowledge of implementation base/pattern architecture		X				X	X	X		X	X	X	X

development centre is based in Germany. The company has three employees educated in computer science: Ole, and two developers. Ole invented and designed the prototype of the systems. He’s the only person in the company that has domain knowledge in the field of telecommunication.

The fifth organisation is GDT, a security software company headquartered in Germany. The company size ranges between 51-200 employees, while 10-11 of them are developers. GDT offers ten security software products for home users and businesses. Those software products have been in development for more than 18 years. We interviewed Hans, a senior software engineer and chief architect at GDT. Hans has been working at GDT more than eight years.

The sixth organisation is CO, a European software publisher founded in 1999 by Belgian Internet pioneers who specialize in virtual office solutions. The organisation is a small company having an approximately ten employees, three of whom are developers. We interviewed Guillaume, a back-end developer and a chief technology officer (CTO) who has been working with the company for more than nine years.

The seventh organisation is XYZ, one of the world’s leading internet companies that provides searching, organises information, and makes it accessible. XYZ has provided dozens of products since the late nineties. The company has approximate 20,000 employees. Although, XYZ is a large company, the size of the development team is kept between 4-6 people. The interaction between teams is the responsibility of architects and product managers. Team members are changed regularly depending on the need of products and projects. The headquarters is located in the USA, but the company has a software development centre in Switzerland where we interviewed Marie, a software engineer educated in computer science. At the time of the interview, she was working on a two-year-old product. Because development processes and practices at XYZ are varied from team to team, the XYZ product shown in Table 2 refers only to the product that Marie is working with.

The eighth and last organisation is DZ, based in Germany. It is one of the world's leading centres for the investigation of the structure of matter. DZ develops, runs, and uses accelerators and detectors for photon science and particle physics. The company ranges between 1,001-5,000 employees who are allocated to various groups. Each group is responsible for its own projects and products. The products are all open-source and are often developed together with other companies and institutes worldwide. We interviewed two employees, i.e. Jan and Matthias. Matthias is a group leader who had proposed his idea to develop the system which was established as a project before Jan, a software engineer, joined the group. The project was to evolve two products that needed to be used together for the system to control cryogenic processes for the colliders. One of the products had been in use for 20 years, and another had been in development for one year. Currently, both products are developed by 2-2.5 developers and are assembled in 15-20 applications.

4.2. The presence of software architecture

This study is based on interviewees' perception on software architecture. We wanted to know about their architectural understanding and how the software architecture is present in the development practice: "*What is your understanding of software architecture?*" The answers were given differently. The term software architecture had been explained using a variety of buzzwords, e.g. a blue-print/skeleton of software, components, design of source code, high-level patterns/abstraction, 4+1 views, structuring, assembling building blocks, stack of technology, layering, dependencies, plug-ins, design patterns, UML diagrams, overall description of the communication of the software, and overview for the development team. The implications of what interviewees understood about software architecture ranged from source code to human activities.

Table 2 summarises the results with respect to software products that interviewees were working with. Product pseudonyms are used to represent the software products as company-based products because of confidential information. The presence of architecture is categorised into *Different dimensions of architecture representation and usage*; *To what detail is the architecture represented?;* and *How is the architecture expressed?*. The cross sign (X) denotes existence of the item with respect to software product names. This is rather coarse information and has to be interpreted together with the qualitative analysis of section 5.

The first category, *different dimensions of architecture representation and usage*, are further categorised into *In which form is the architecture presented in the process?*; *How is the architecture* *represented?*; *How the architecture representation is used?*; and *How can the architecture representation be updated?*. For example, the architecture for EW products is presented in the form of *in somebody's head*, and *documented in folder, binder, or internet*, but is not *readily available in the workspace*. The architecture for EW products is represented in *text*, *source code*, *boxes and arrows*, *class and package diagrams*, *Architecture Description Language (ADL)*, and provides *different views*. The architecture representation for *EW products* is used in *design*, *communication between developers*, *communication about changes*, *communication when designing new features*, *communication for bug fixing*, *as feedback for on going implementation*, *distribution of work and responsibility*, and *generation of diagrams*. The architecture representation for EW products is updated, *regularly controlled*, *related to an overall plan and release plan*, or *only when problem occurs*. The many marks in EW's column indicates an elaborated practice which might have been due to the cooperation with the local university.

The next category is *To what detail is the architecture represented?* containing five levels: *overall*, *classes*, *styles*, *patterns*, *design patterns*, and *various views*. For example, the architectures of ABC product#1-5 are all detailed at overall level. The architecture of ABC product#1 is detailed at the levels of *overall*, *classes*, *styles*, *patterns*, and *various views*, but not *design patterns*. The architectures of ABC Product#3-4 are detailed at the levels of *overall* and *classes*. The architecture of ABC Product#5 is detailed at the levels of *overall*, *classes*, and *patterns*.

The last category is *How is the architecture expressed?* that is further categorised into *implicitly*, *explicitly*, and *requiring substantial knowledge of implementation base/pattern architecture*. For example, the architectures for OMD products are *explicitly* expressed using *diagram* and *requiring substantial knowledge of implementation base/pattern architecture*.

Based on Table 2, the architecture is mostly presented in the process as a form *in somebody's head*. All our interviewees report that the architecture is represented in the *source code*, but architecture description languages (*ADL*s) are hardly ever used to represent the architecture. The architecture representation is commonly used for *design*, *communication between developers*, *communication about changes*, *communication when designing new features*, and *distribution* *of work and responsibility*. Updates of architecture representation are almost equally distributed between *never* and *regularly controlled*. However, the updates rarely *relates to overall plan and release plan*. Most of the interviewees confirmed that the architectures are represented at the *overall* detail; only a few addressed *styles* and *various views*. The architectures are almost equally expressed *implicitly* in *source code*, *explicitly* in *diagram* and almost always *requiring substantial knowledge of implementation base/pattern architecture*. However, the architectures are rarely expressed *explicitly* in *textual description*.

5. Analysis of interviews

Our interview study focused on the real life application of architecture in daily development practice. Based on our interview data, we drew a diagram, as shown in Figure 1, to organise our data, concepts, categories, sub-categories, and the relationship between them. We continued comparisons of categories against actual data in order to substantiate possible explanations which resulted in the following sub-sections: Sub-section 5.1 begins the analysis with target groups of architecture and their understanding levels in the architecture; Sub-section 5.2 is documentation for the architecture; Sub-section 5.3 explains how newcomers learn the architecture of software products; Sub-section 5.4 points out the role of architect(s); Sub-section 5.5 shows communication channels that developers use for updating information about changes in their software products; Sub-section 5.6 addresses the architecture with respect to evolvability and changes; and Sub-section 5.7 presents architectural problems as addressed by our interviewees.

In the presentation, we use citations from the interviews to illustrate and support our analysis. These citations keep as much as possible to the original wording. We minimally edited them to for readability and Grammar.

5.1. Architecture: who needs it and at what level?

Throughout the software life cycle, development team members carry on their tasks depending upon their roles. Different team members use architecture in different ways in order to collaborate with others. The interviewees distinguished three groups, i.e. newcomers, developers, and chief architects. The newcomers need architecture as a springboard to understand the software that they are going to develop. The developers share and accumulate architectural knowledge, in particular, the part of architecture that they are responsible on a daily basis. The chief architect orchestrates all architectural activities based on their architectural knowledge.

The different roles refer to the architecture on different levels of abstraction. These levels define a protocol on how to discuss and understand the architecture. Marie from XYZ company said: *"Usually, we talk to each other verbally [face-to-face], and we can imagine [understand] because we know the code basis. When we do it on the team, we never go to class level. Otherwise, nothing will be done. We define the interface level, i.e. how do we talk to each other*." But not everyone will be able to look from a high-level abstraction point-of-view. Guillaume, the CTO from CO company, told us when he reviewed his colleague's work, the code worked fine, but it was not easy to understand. The main challenge was that it required a lot of abstraction to interpret. "*You don't have to tell how the code works in the [function name], but what it does. For example, yesterday, he [Guillaume's colleague] wrote a function 'setStyle'. What the function does is to change the style. But that is very low-level interpretation. In fact, at the high-level it is to add background to the menu and the exact name would be 'setBackground', not 'setStyle' ... ..*" The difference between levels of abstraction became visible again when we asked Jan and Matthias from DZ to draw and explain their architecture. They worked on the same product, but Matthias explained the architecture as the overall picture, while Jan explained what he was responsible for on the daily basis.

5.2. Documentation

Forms of documentation and how it was used was subject to each of the interviews. The code base presented throughout the interviews is regarded as the best and only up-to-date representation of de-facto architecture, whereas independent documentation is regarded as problematic because it is outdated quickly.

5.2.1. Code base as actual documentation

Source code is seen as ‘the’ actual documentation while the other kinds of documentation are informally produced to support situated discussion. However, UML diagrams are produced in small scale mostly by need, for example, to get feedback from other developers before implementation of a large component. Direct discussion with developers is more efficient. Many integrated development environments (IDEs) support synchronisation between UML and source code, however, the developers feel comfortable to start with programming. The developers have the impression that understanding and becoming familiar with UML diagrams takes longer than looking into source code.

A common problem is a lack of documentation on the overview of a system, in particular a design rationale, and the description of the main interfaces or functions. A few documents are provided for newcomers to become familiar with architecture. Newcomers feel that comprehending systems from documents only, is hopeless, so they as well prefer to start with programming.

When the source code becomes the actual documentation, the naming of classes, methods, or interfaces is extremely crucial for on-going development. If the name is on the correct level of abstraction, it facilitates the other developers to understand the concept behind the name. The citation above by Guillaume shows that architects are well aware of this need and take care to implement it. Apart from the on-going development, shared distributed development or end-users development also gain this benefit.

5.2.2. The absence of a document

Our interviewees give several reasons for the absence of explicit documentation. Some software products have been used more than 20 years. When the products were first developed, the main purpose of development was to solve domain problems for a short time, thus, there was no effort on architectural documentation. When developers started with a small feature, they neglected to document, so when that feature grew bigger, documentation hardly ever existed.

Though a document might have been created, the effort of keeping the document up to date leads to maintenance neglect. Developers have the responsibility to document what they have been programming, but they are aware that their documentation will soon be out of date. Marie said: "*Documentation is like [...] cleaning. You have to clean regularly, but it will get dirty again. When we document, we know that the documentation will be out of date soon*." Maintenance and updating documents is a boring task for developers. The architecture document often has a simple notation or diagram using boxes and arrows, so it's not convenient to update the diagram. Therefore, documentation diverges from what is actually present in the source code.

In a complex and specialised domain area, e.g. hydraulic simulation, domain expertise is strongly required for developing a software product. Often, a developer is a domain expert rather than a software expert because it is a big task for the software expert to get familiar with the domain, so documentation is often neglected by domain experts. This in turn becomes a problem. Our interviewees reported that even a developer with domain expertise could spend up to a year to fully understand and implement a new functionality in a software product. The main reasons are not only the complexity of software products building on top of a stack of technology, but also the unavailability of architecture documents. The architecture exists in somebody's head rather than a written document. When a developer or an architect doesn't document the current architecture before leaving a company, it causes problems for other developers who follow that architecture.

5.3. Architecture knowledge acquisition: how newcomers learn the architecture

A well-attuned team might not have any problems with informal architectural practices, so we asked specifically how new developers are introduced to software architecture. All our interviewees reported on their informal knowledge-sharing practices. In this sub-section, we show how architectural knowledge is acquired and developed by team members, discussed with a chief architect, intermixed with programming, and learning by experience.

5.3.1. Discussion with a chief architect

Throughout a software's life cycle, a chief architect discusses with developers in order to update the progress of the tasks, and realise the changes in the architecture. Since tasks are clearly distributed based on architecture, each developer is responsible for his task, for example, making new features or adding some functionality. The chief architect's discussions with developers ensure that they understand the tasks before they begin implementation. The chief architect often draws boxes-and-arrows or UML-like diagrams on a piece of paper, or whiteboard, or makes a Power Point presentation of a software prototype and its architecture. If the chief architect's explanation is not precise, bad design decisions could result. In order to avoid that situation, one of ABC senior developers explained how he transfers architectural knowledge to the ABC off-shore developers: "*We go to China and explain this [diagram] to them. Then we show them how things [architecture] are now and explain [...] what we want to add to this, [the] new feature to put in….*"

The chief architect synchronises architectural knowledge with other developers by discussion, in particular, team members situated at the same physical location. The discussion happens in weekly meetings or daily conversations that take place spontaneously in communal areas, like kitchens, canteens, or coffee corners. The chief architect has the most up-to-date architectural knowledge, but that knowledge is hardly ever documented. In order to get that knowledge, one has to update from the chief architect. When we asked Neeraj, a junior developer at OMD, about a certain architectural style and patterns currently used for OMD software products, he said: "*You have to ask our system architects. I'm not able to answer that.*"

5.3.2. Intermixed with programming

Training with a chief architect or a senior developer is a springboard for newcomers to understand architecture. Based on our interviewee's experience, a common training technique is pair-programming, where two developers program together at a workstation. In the vast majority of cases, the newcomer is assigned to implement a simple task. The chief architect, or the senior developers, sit down with the newcomers and allow them to ask questions. The newcomers get an overview and design rationale of the software product. They develop a 'feeling' – as one of out interviewees expressed it – of how the software product works, how to implement using provided tools, and how the repository is organised. The newcomers look into packages, files, and source code, and at the same time, begin programming, so they gradually learn how the software is architected. Later on, the newcomers become developers who are responsible for the architecture of his or her sub-systems. This intermix process of architecting and programming facilitates both architecture acquisition and work progress. However, the chief architect or the senior developers need to contribute his or her time to educate each newcomer.

5.3.3. Learning by doing

A software product can be developed on top of a third party software product. Often, the third party architecture is not fully transparent, nor does it provide sufficient architectural documentation to explain how the system is implemented or how it can attach a new functionality. Every time the developers and the chief architect begin developing on top of a new third-party software product, they feel like they are 'opening Pandora's box.' Moreover, changes are hardly controlled. When the third party releases a new version, the only way to understand that new architecture is to look directly in the source code. It is always a time-consuming and painful process.

5.4. The role of a chief architect

The analysis so far points to and underlines the importance of the software architect acting on what we have begun to call a 'walking architecture.' Not all companies have a 'chief architect'. However, all products have a person or group of people acting in that role, even though their title might be different, like chief technology officer (CTO), senior developer, product manager, project leader, or system architect. Chief architects have – explicitly, or due to the recognition of their expertise – the responsibility for designing and updating the architecture throughout the software life-cycle. The chief architect informs and updates the developers regarding architectural changes on a daily or weekly basis. In their formal and informal meetings, the developers also update the chief architects about architectural issues in the parts of the program for which they are responsible. The chief architect sometimes creates documents containing a few diagrams to give a good view of the software, however, most developers still prefer talking directly to the chief architect. The developers usually ask about relevant parts of the software that are changing, and even publicly-kept architectural documents. Still, the most updated version of architecture is 'stored' in the head of the chief architect. This is also the understanding of the chief architects themselves, as it becomes apparent in a question put to Hans, the chief architect from GDT, on how he knows about the current or de facto architecture. "*I've worked in the company for eight years. Most of architectures are my architecture*s."

As our interviewees emphasize, a good chief architect has both expertise in software engineering, and the domain. This sub-section categorises three main roles of the chief architect, i.e. controlling and communicating architecture within a development team, and interfacing to outward.

5.4.1. Controlling and communicating architecture within a development team

A chief architect is responsible for most design decisions. In initial design discussions, the chief architect sometimes brainstorms with domain and software experts before designing the architecture. This discussion covers data type, quality attributes, design patterns, and platforms for the architecture. Most of the design architectures have clear interfaces and low dependency between components. The design architecture is often used for distributing developmental tasks and defining social protocols that aim at using the architecture as a coordination mechanism. Examples for such protocols are: when changing an interface the programmer must contact the relevant developer; the chief architect synchronises the tasks by collecting, reviewing and accepting everything before checking changes and documenting requirements for the next release; and the chief architect schedules meetings or workshops with developers when the architecture needs to be updated.

During the implementation, some types of problems cannot be resolved by tools automatically, e.g. naming of a function at the right-level of abstraction. Thus, code review is often part of the tasks of chief architects. When they find a problem in the implementation, the chief architects will talk directly to developers and motivate them to resolve the problem.

Chief architects often train newcomers by assigning simple tasks, e.g. implementing a new component. They know where to add these new components or functionalities without endangering the architecture. When they have sufficient knowledge, the chief architect will assign developers to work on critical parts.

5.4.2. Updating the 'walking architecture'

Maybe because we did not ask explicitly, the interviewees did not always emphasise the above understanding of the mentioned practices, in that they may also serve the purpose of updating the chief architect's knowledge of architectural issues that might lead to reconsidering the architecture itself. This became visible in two ways within our interview material; the communication with developers was talked about as a two-way communication, and the problems that arose when the feedback channel did not exist.

Marie from XYZ answered the question of how their team discussed architectural issues: "*When it becomes larger, especially [if] it affects a whole sub-team, or the other part/team, or [we need a] sanity check, we set up [a] meeting or talk informally with the people who care about the affect and need to know.*" Or as one of the project leaders at ABC said: "*Generally, it's [a] very informal way, [talking] between colleagues that know about this thing.*" On the question of how developers get to know about relevant changes in the architecture, another of the ABC architects answered: "*Hopefully the one making the changes tell other people.*"

These informal update becomes problematic when the development becomes distributed. One of ABC senior developers reported: "*Sometimes they do not. […] Normally, when developers were in Prague, most things [were] developed there, [so] they knew what [was] going on and we [had] weekly meeting with them, talking about different things, and what different people [had] been doing. […] Now,* *it's not the same [as] what we do in China. So, it's more difficult, now. Developers do not know anymore what changes in the application. [...] It's difficult.*" One of our interviewees reported: "*Sometimes we have people implementing core components that destroy other component. Not so much within the engine group, because we have only two people. But the other, especially Singapore or Shanghai groups that did some core components change, because there [was] no documentation up there. They didn't know that the components [failed] because they [relied] on special functionality.*"

Some of our interviewees reported about explicit measures to stay up-to-date with the architecturally relevant changes to the software product. A protocol might have been established that developers must inform the chief architect before changing central parts, core components, or data structures.

The integrated development environment can indicate architecture violations if set up in the right way. Also, nightly builds can indicate when new code breaks an interface. In some companies, the chief architect reviews changes to the code and based on the reviews, discusses changes with the developers.

Guillaume from CO, keeps up-to-date with changes in the common parts of the software through regularly reading the source code and common Wiki. "*In the Wiki, every change in the software is documented, not with a lot of detail. ... In the trunk [of the CVS], we document every commit. ... I take care of [reading] source code, Wiki and the commit.*"

5.4.3. Interfacing to outward

Chief architects reported that they need to interface with people working outside of the development team, e.g. people gathering requirements and working with other related products, clients, or end-users. In order to utilise the design architecture, chief architects have to ensure that agreement with people working outside the development team have been made. In a conflict situation, chief architects need to negotiate and compromise with those outside sources.

Chief architects are aware that feedback from outside professionalises the development. They often discuss expectations of implementation with their clients before conversations with developers. They go back and collect feedback from the clients or the end-users before the next release. If chief architects have no direct contact with clients or end-users, they have discussions with sales and marketing people. The feedback from the clients or marketing people will help the chief architects get a correct understanding of the requirements. Based on this understanding, they prioritise and delegate the requirements for the next release.

Due to business or organisational reasons, related software products might be developed in different units. In some cases, more than one team together develops a software product. In such situations, chief architects needs to coordinate with other development teams in the same company or from contract companies. In software product lines, changes in one product may cause malfunctions in other products. Thus, chief architects must take heed of the changes. If problems occur, it is their task to find solutions. The solutions vary from collaboration to architecting. Examples are as follows: the chief architect communicates with another development team on the changes overall effect; the chief architect develops interfaces (e.g., API) to express a common concept of another product; or the chief architect designs the architecture in a way that accommodates the interest of both teams. Guillaume, a chief technology officer at CO, told us how he handled recent changes: "*Last week, XYZ company published an API to access the address book [of XYZ product]. There [was] a request to implement a mechanism to import the address book from XYZ to CO. XYZ implemented most of APIs under a common umbrella [XYZ product]. I think I have to take care; I develop[ed an] interface in [a] generic way in order to express the concept of [XYZ product]*."

5.5. Communication about changes

Team members need to be updated about changes. In this sub-section, we categorise communication from the current software development practice that covers both human interaction and support tools. Each practice presented below is ranked from the most commonly used to rarely used. Note that each interviewee reports on more than one practice.

Verbal communication or *face-to-face communication* (i.e., free-form dialogue, explanation, and discussion) with colleagues is the most common practice. It is used by all interviewees and seems to be the simplest way to update them about architectural changes. Just another quote from our interviews complementing the many above: "*We don't have state diagrams and seldom use sequence diagrams. But we need knowledge about what [method is] to be called first or second. That's not explicitly stated. It is just something that we make use of by asking people that know about this typical sequence of calling.*"

Meeting. A development team must meet regularly where each developer goes through all the tasks and updates what the other colleagues are doing, so the team members can synchronise their understanding of the architecture. In some distributed development projects, a whole development team assembles for a longer 'coding camp' meeting at the same location in order to brainstorm about new designs, or to finalise a new release.

Nightly builds and testing. Nightly build mechanisms notify developers the next morning if the changes checked in the day before had affected other parts of the software, for instance, breaking interfaces or violating the design rules.

Email, mailing list, and instant messaging are used spontaneously by team members to send messages (e.g., "*By the way, you broke our code!*"), or inform the other members within a team, or between teams, about changes. Sometimes, messages are automatically sent from an IDE or a support tool.

Concurrent Versions System (CVS) and Subversion repository. In some cases, everyone in a team has their own branch on repository to work on as a sand-box. Later, they carefully merge all the changes in a common branch or a trunk in order to rebuild the software. When a developer commits changes in the source code into the trunk, the CVS repository automatically sends an email to the other developers.

Rich IDE. Some IDE provides multi-disciplined team members with an integrated set of tools for architecture, design, development and testing of applications. The IDE can report problems in architecture and do quality assurance. Ole from ARG told us how his developers knew the relevant parts of the software were changing: "*[IDE name] tell us which part of the architecture has problems. ... [IDE name] is a golden gun. It is a very complicated environment ... It can do much for software engineering.*"

The main advantage of the integration is to handle the changes within a monolithic tool. Guillaume from CO supported this with: "*I* *can modify code and the code is still consistent for all applications ... It is quite easy to handle.*"

Code review. Changes in the source code are sometimes reviewed by chief architects before one can commit to repository. They correct mistakes in the source code and improve the quality of software while doing code review, then often discuss the changes and rationale with developers.

Wiki. Every change in the software can be documented in a Wiki. Though the documentation is not fully detailed, everybody is aware of the changes and can use the Wiki to inform about the ones he introduced. (See also section 5.4.2.) Only few interviewees report the use Wikis to update or inform team members about changes to the source code or the architecture. However Wikis are sometimes used for collecting ideas and requirements from users and developers.

5.6. Evolution and changes

When the original architecture had first been established, people had no intention of ever changing it. However, changes initiated by use and business contexts of software products resulted in new requirements which in turn affected the architecture. Typical examples include: a user request for some functionality that could benefit other users; an intuition from a developer who might even use the software himself triggers changes; and marketing strategy or competition. From all organisations, our interviewees confirmed that chief architects have to be involved in or responsible for all changes, in particular, architectural changes. Chief architects need to satisfy the change requests and existing architecture in order to reduce the effect on the architecture. They might decide to add new components or functionalities. If re-design of the whole architecture is the only solution, they could suggest creating a new software product.

Regarding changes implied in the development, protocols might be put in place to support information and knowledge sharing. For instance, developers are not allowed to change a common part, core component, or data structure without informing the chief architect; the developers should synchronise their changes with their colleague; or developments have to be done based on the latest version.

Although chief architects are initially responsible for establishing architecture and taking care of changes, it is difficult to keep track when architecture evolves over long periods of time. Changes in source code affect the other parts of a software product, for example, changing a common part causes a software malfunction. One of ABC senior developers complained: "*On the entire ABC product line, if you change something on the core components, you may destroy 95% of the component here…. It's difficult to know exactly what component you are touching by changing the code.*" Furthermore, the changes sometimes have effect beyond a company's boundary. Ole, a managing director at ARG, told us about customising a third party product: "*When they [the third party product developing companies] make changes on the architecture, we know it by the malfunction on our software, not by documentation.*"

5.7. Problems

Our interviewees also talked about problems in their architectural practice. The problems address technical infrastructure as well as co-operational aspects of software development. The common problems in the technical context are changes in technical infrastructure, framework, or standard that a software product builds upon, e.g. changing virus scanners to support Unicode base, or changing global unify identification mechanisms for CRM systems.

With respect to co-operational aspects, some software design/developmental approaches are likely to obstruct day-to-day development practices and hinder collaboration. Furthermore, a lack of awareness, a lack of domain or software engineering expertise, or the loss of architecture knowledge often causes architectural problems.

Gaëtan, a senior developer at EW, reported from his daily practice: "*we have a problem with the model; it is a binary file, not text file. So it means that if I want to change a part of the model, and, at the same time, the other developer wants to modify another part. It is just one file. So one of the two guys can make modification, the other cannot touch anything. Once one finishe[s] and commit[s] the file, another can check-in. It is not easy to modify the design with more than one [person] at a time. For source code is different; the code is in plenty of files. People can work on separate sets of the files. When people work* *on the same file, we can 'make diff' between elements and see [the code] that a guy works on this part [...] and merge [the] changes, or the changes can be merged automatically because it doesn't touch the same part. It is easier to work [in a] collaborative way with code. But with this [the model] is not possible.*" It is reasonable to claim that, apart from being 'the' actual document, code basis supports collaborative development better than the model.

Although many existing tools and practices are used for informing changes and controlling evolution (e.g., nightly builds, unit test, or regression test), these tools do not resolve all the problems. Lacking information about changes as reported in section 5.4.2 is common, especially in distributed software. Although documentation problems are addressed here, creating a document might not be the best, or the only solution. Hans, a chief architect at GDT, said: "*If it [the document] would be a good view on the software, I will do it. ...We don't need overview for every class.*" Therefore, giving and controlling information about changes should be done at the right level.

One of ABC senior developers stated: "*This kind of dependency graph would tell them we have to tell someone about something, that we [need to] make changes. It helps people be aware that if they change here [a component it] may affect [something] elsewhere. Otherwise, it will be difficult to see by [yourself].*"

Finding the right people and keeping them is a challenge in many software developing companies. "*One year ago, unfortunately, our excellent developer left our company. It is very hard to find [new] developers. We are looking everywhere,*" Guillaume, a chief technology officer at CO complained.

Developing software products need both domain and software engineering expertise, but they rarely come together. Ole, a managing director of ARG, addressed his biggest concern: "*our developers do not know anything about telecommunication via telephone, although they use it. ... On the other hand, [third party product] developers understand very well about telecommunication and techniques of voice over IP, or something like that, but they don't know anything about software factories, processes or design patterns. They know nothing about the architecture and development cycle.*"

If newcomers or developers have only software engineering expertise, they will need time to acquire sufficient domain expertise. If the developers have only domain expertise, they will need time to get software engineering expertise. When developers acquired both, their expertise becomes an important asset for companies. However, companies cannot always hold onto their personnel. Losing a central developer or chief architect can result in the failure of a software product. One of ABC senior developers said: "*the software products are going to the dying phase or dead-code when the key programmer left. ... We don't know how to write them.*"

6. Discussion

This empirical study focuses on the development of software products. The practices revealed in this study might be different from the contract development [45] or the development of high-integrity systems [31]. Software products constantly evolve. Changes in the software product must be handled consciously in order to prevent dead-end development. Through our interviews it became clear that the main issue is not to implement the design architecture, but to maintain the architecture when evolving the software in a viable state, so it can support future requirements and innovations. In the discussion now, we highlight the aspects that we consider relevant for developing support for architectural practices for software product development. The importance of the 'walking architecture', 'good reasons for bad documentation' indicate the need to develop social protocols fitting with local practices when introducing architecture representations and documentation, and we finally propose a means to promote architecture awareness.

6.1. Architecture awareness is achieved through Walking Architecture practices

The analysis indicates that product development teams depend on chief architects or group of architects who act as what we started to refer to as the 'walking architecture' for communicating the architecture to the developers, and in turn communicate problems which might become architectural issues to the software architect. The walking architecture takes most, if not all, design decisions and solves architectural problems throughout on-going development. Architectural issues arise from inside as well as outside the development team, cover technical and social aspects of software development, and require domain, as well as software engineering expertise. In order to solve these issues, the chief architect interacts with technical and business people, establishes tools and practices, and recruits or trains team members for that expertise, etc. Because architecturing is not just only a matter of technical design, but also of juggling the social contexts of software development that make it almost unable to automate [71].

Our analysis both confirms the importance of inward and outward interaction as part of the role of the software architect [34], and deepens the understanding of the importance of this interaction. In the interviews, the rational behind these practices becomes visible. On the one hand, developers need up-to-date knowledge about the architecture, here and now. The architect can explain the structure of the software in relationship to the problem at hand. On the other hand, the chief architect may stay in contact with the development of the source code and become aware of potential issues. The emphasis on face-to-face communications provides a strong indication that whatever methods and tools software engineering research proposes needs to be aligned with the practices of knowledge-sharing by, and with, the walking architecture.

6.2. Good reasons for bad documentation

A lack of up-to-date architecture documentation is problematic according to our interviewees, and software architecture researchers. If the lack of documentation phenomenon is so widespread, one might suspect ‘good reasons’ for ‘bad documentation’ (See also [30].)

As presented in section 2, architecture research emphasises written representation (e.g., formal notations or documentation) and codified knowledge. Written representation describing software architecture might suit a researcher’s practice rather than a chief architect’s practice. Based on our empirical evidence, the architecture almost always exists in somebody’s head. Architecture knowledge management can be described as socialisation-heavy. Face-to-face communication is the-state-of-the-practice of architectural knowledge management. For example, we often overhear a team member say to his/her colleague: “*I know you worked on this component, please tell me about it,*” or “*The best person to ask is the architect.*” Team members are used to conversing with a chief architect about their work. Through these discussions, chief architects not only educate and inform developers, but also take heed of changes that may cause problems to the architecture later on. They converse with the team members in order to find a solution for architectural problem. Given these practices, the absence of documents is not a risk for software companies. On the contrary, if documentation was successfully established even in parts instead of the aforementioned practice, the chief architect would lose track of what is going on in the architecture. As a consequence, nobody could maintain the architecture anymore, which may in turn result in serious problems.

If a shared form of documentation is established, social protocols need to be established, as well, to make sure that the walking architecture learns about developments in the code and potential architectural issues. One example of such a social protocol is the practice of regularly reading the common wiki-based documentation, the checked in source code, and the CVS information in order to keep up-to-date with the changes to the code.

6.3. How to promote architecture awareness

Practitioners and researchers agree on the importance of software architecture being part of everyday software development in order to enhance quality attributes [47], in particular evolvability [72]. Many software companies have successfully evolved their products even though they hardly ever emphasise explicit architectural documentation, or keep architecture documents up-to-date. However, source code is a reification of design. Developers and architects are well aware of the architectural structures: software developers know when to change the source code, where to change it, who to ask, who to inform, etc. Architecture is alive with a walking architecture.

Tools and methods for promoting architecture awareness should support this practice, rather than establishing a diverging approach. Two promising examples of how to do this are as follow:

One of our interviewees reports on his projects sharing and cooperatively maintaining the architecture knowledge in the form of a common wiki (See section 5.4.2.). Documentation in Wiki and CVS, albeit not very detailed, can communicate changes to team members, and especially the chief architect. The documentation in Wiki and CVS becomes one way of communicating about changes continuously that does not hinder, but supports the chief architect. The chief architect can then read through the changes and be aware of anything that might affect the architecture.

A similar tool is proposed by Solis et al. [65]. Personal communication (Muhammad Ali Babar) on the usage of this tool indicates that a social protocol similar to the one reported above, evolved around its usage.

As part of the research with a product developing company, Unphon [69] presented the introduction of a 'build hierarchy' matching the static architecture as a technique to give developers continuous feedback about whether their code complied with the design architecture. This architecture-based built hierarchy can be implemented as part of the integrated development environment (IDE), or the nightly build infrastructure. That way, the build hierarchy supports on-going architecting through architectural compliance checking between design architecture and code architecture. If implemented as part of the IDE, developers can be informed with every compile command whether their changes affect the other parts of the software or break the architecture. If changes in the source code break the design architecture, the developers need to revise the changes or discuss it with the chief architects and their colleagues. Through the discussion, the chief architects are updated about development problems that might become architectural issues.

7. Conclusions

This article began with postulating the question, what architecture practices do software product developing companies apply to keep their products alive, sometimes over several decades. The study presented in this paper shows that architecture practices emphasise face-to-face communication rather than the codification in documents. The analysis emphasises the importance of a chief architect acting as a walking architecture who is responsible for maintaining and evolving the software products' architecture. Through face-to-face communication with developers, as part of the everyday development, the chief architect communicates the software architecture in a form most suited to help the developers with problems at hand, and at the same time, becomes aware of potential architectural problems.

Other research has promoted documentation as a recommended practice for development teams. Based on our analysis, we hesitate to join this chorus. A document quickly becomes outdated if not continuously maintained. Moreover, the documentation could disrupt the practice of the walking architecture: if reading the document replaced discussions, the chief architect would not be informed about potential problems in a timely manner. In the long run, being uniformed could result in changes to the overall architecture that may conflict with the needs and constraints of different modules.

Applying the notions of awareness and social protocols as a base for sharing information about a cooperatively achieved task allowed us to take another approach to tools and methods supporting software architecture. If document and tools are devised, they need to be fitted into everyday developmental practices, and require a change in the social protocol around architecting. As examples, we discussed the usage of a common Wiki from our field material, and using the build hierarchy as a reification of the design architecture based on related research. With this result, the article confirms the importance of taking cooperative aspects into account when devising solutions for seemingly technical problems.

ACKNOWLEDGEMENTS.

We kindly thank all interviewees for participating in this study. We should like to extend our gratitude to Dr. Wolf-Gideon Bleek for his inspiration and help in setting up this study.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6 (3): 213–249, 1997. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/258077.258078.

[2] R. B. Allen and D. Garlan. A Formal Approach to Software Architecture. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 134–141, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co. ISBN 0-444-89747-X.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[4] L. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15 (1): 225–252, 1976.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, Reading, Massachusetts etc., September 1998. ISBN 0201571684. URL http://www.amazon.com/-exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201571684.

[6] J. Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-67494-7.

[7] F. Brooks and K. Iverson. *Automatic Data Processing (System 360 Edition)*. John Wiley, 1969.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[9] K. Czarnecki, U. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000. ISBN 0201309777.

[10] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the Wild: Why Communication Breakdowns Occur. In *Proceedings of the international Conference on Global Software Engineering*, pages 81–90. ICGSE, IEEE Computer Society, Washington DC., August 27 - 30 2007. doi: http://dx.doi.org/10.1109/ICGSE.2007.13.

[11] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: software source code as a social and technical artifact. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 197–206, New York, NY, USA, 2005. ACM. ISBN 1-59593-223-2. doi: http://doi.acm.org/10.1145/1099203.1099239.

[12] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In *CSCW '04: Proceedings of the 2004*

ACM conference on Computer supported cooperative work, pages 63–71, New York, NY, USA, 2004. ACM. ISBN 1-58113-810-5. doi: http://doi.acm.org/10.1145/1031607.1031620.

[13] T. Dingsøyr and R. Conradi. A survey of case studies of the use of knowledge management in software engineering. *International Journal of Software Engineering and Knowledge Engineering*, 2 (1): 391–414, 2002.

[14] K. Dunsire, T. O'Neill, M. Denford, and J. Leaney. The ABACUS Architectural Approach to Computer-Based System and Enterprise Evolution. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 62–69, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2308-0. doi: http://dx.doi.org/10.1109/ECBS.2005.66.

[15] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis &Design Language (AADL) a standard for engineering performance critical systems. In *Proc. IEEE Computer Aided Control System Design IEEE International Conference on Control Applications IEEE International Symposium on Intelligent Control*, pages 1206–1211, 4–6 Oct. 2006. doi: 10.1109/CACSD-CCA-ISIC.2006.4776814.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] M. Fowler. Who needs an architect? *IEEE Software*, 20 (5): 11–13, 2003. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231144.

[18] D. Garlan and M. Shaw. An Introduction to Software Architecture. Technical report, Pittsburgh, PA, USA, 1994. URL http://portal.acm.org/-citation.cfm?id=865128.

[19] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[20] E. Gasparis, J. Nicholson, and A. H. Eden. LePUS3: An Object-Oriented Design Description Language. In *Diagrams '08: Proceedings of the 5th international conference on Diagrammatic Representation and Inference*, pages 364–367, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87729-5. doi: http://dx.doi.org/10.1007/978-3-540-87730-1_37.

[21] E. M. Gerson and S. L. Star. Analyzing due process in the workplace. *ACM Trans. Inf. Syst.*, 4 (3): 257–270, 1986. ISSN 1046-8188. doi: http://doi.acm.org/10.1145/214427.214431.

[22] B. G. Glaser and A. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, Chicago, 1967.

[23] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004. ISBN 0471202843.

[24] C. Gutwin and S. Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work*, 11 (3): 411–446, November 2002. doi: http://dx.doi.org/10.1023/A:1021271517844.

[25] C. Hansson, Y. Dittrich, and D. Randall. Agile Processes Enhancing User Participation for Small Providers of Off-the-Shelf Software. In *Extreme Programming and Agile Processes in Software Engineering. Proceedings of the 5th International Conference, XP 2004*, Garmisch-Partenkirchen, Germany, June 6-10 2004.

[26] C. Hansson, Y. Dittrich, B. Gustafsson, and S.Zarnak. How Agile are Industrial Software Development Practices? *Journal of Systems and Software*, 79: 1295–1311, 2006.

[27] C. Hansson, Y. Dittrich, B. Gustafsson, and S. Zarnak. How agile are industrial software development practices? *J. Syst. Softw.*, 79 (9): 1295–1311, 2006. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2005.12.020.

[28] N. B. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Software*, 24 (4): 38–45, 2007. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2007.124.

[29] C. Heath and P. Luff. Collaboration and Control: Crisis management and multimedia technology in London Underground Line Control Rooms. *Computer Supported Cooperative Work (CSCW)*, 1 (1-2): 69–94, March 1992. doi: 10.1007/BF00752451.

[30] C. Heath and P. Luff. Documents and professional practice: "bad" organisational reasons for "good" clinical records. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 354–363, New York, NY, USA, 1996. ACM. ISBN 0-89791-765-0. doi: http://doi.acm.org/10.1145/240080.240342.

[31] M. G. Hinchey and J. P. Bowen. *High-Integrity System Specification and Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540762264.

[32] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-32571-3.

[33] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12 (6): 42–50, 1995. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/52.469759.

[34] P. Kruchten. The Architects—The Software Architecture Team. In P. Donohoe, editor, *Software architecture: TC2 first Working IFIP*

Conference on Software Architecture (WICSA1), pages 565–583, San Antonio, Texas, USA, February 1999. Kluwer Academic.

[35] P. Kruchten. Controversy Corner: What do software architects really do? *J. Syst. Softw.*, 81 (12): 2413–2416, 2008. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2008.08.025.

[36] P. Lago, P. Avgeriou, R. Capilla, and P. Kruchten. Wishes and boundaries for a software architecture knowledge community. *Software Architecture, Working IEEE/IFIP Conference on*, 0: 271–274, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/WICSA.2008.25.

[37] M. Lehman. On Understanding Law, Evolution, and Conservation in the Large-Program Life Cycle. *Systems and Software*, 1 (3): 213–231, 1980.

[38] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X. URL http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/556.pdf.

[39] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. ISBN 0201042053.

[40] B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Commun. ACM*, 24 (11): 763–769, 1981. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358790.358796.

[41] D. C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Technical report, Stanford, CA, USA, 1996.

[42] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21 (9): 717–734, 1995. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.464548.

[43] N. H. Madhavji, J. Fernandez-Ramil, and D. Perry. *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, 2006. ISBN 0470871806.

[44] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: http://doi.acm.org/10.1145/302405.302410.

[45] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002. ISBN 0-201-63460-0.

[46] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of*

software engineering, pages 18–28, New York, NY, USA, 1995. ACM. ISBN 0-89791-716-2. doi: http://doi.acm.org/10.1145/222124.222136.

[47] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Inc., 2008.

[48] P. Naur. Programming as Theory Building. *Microprocessing and Microprogramming*, 15: 253–261, 1985.

[49] I. Nonaka. A Dynamic Theory of Organizational Knowledge Creation. *Organization Science*, 5 (1): 14–37, Feb. 1994.

[50] I. Nonaka. The Knowledge-Creating Company. In *Harvard Business Review on Knowledge Management*. Harvard Business School Publishing, Boston, 1998.

[51] D. K. Padgett. *Qualitative methods in social work research*. SAGE Publications, 2nd edition, 2008.

[52] D. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, North Holland, 1971.

[53] D. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15 (12): 1053–1058, 1972. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361598.361623.

[54] D. Parnas. On a ’Buzzword’: Hierarchical Structure. In *Proceedings of the 1974 IFIP Congress*. Kluwer, 1974.

[55] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2 (1), 1976.

[56] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng.*, 12 (2): 251–257, Feb. 1986.

[57] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005. ISBN 3540243720. URL http://www.amazon.com/exec/obidos/-redirect?tag=citeulike07-20&path=ASIN/3540243720.

[58] C. Robson. *Real world research: a resource for social scientists and practitioner-researchers*. Blackwell publishing, UK, second edition, 2002.

[59] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt. Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Softw. Engg.*, 8 (4): 351–365, 2003. ISSN 1382-3256. doi: http://dx.doi.org/10.1023/A:1025368318006.

[60] K. Schmidt. The Problem with ’Awareness’: Introductory Remarks on ’Awareness in CSCW’. *Computer Supported Cooperative Work*, 11 (3): 285–298, 2002. ISSN 0925-9724. doi: http://dx.doi.org/10.1023/A:1021272909573.

[61] K. Schmidt and C. Simone. Coordination mechanisms: towards a conceptual foundation of cscw systems design. *Comput. Supported Coop. Work*, 5 (2-3): 155–200, 1996. ISSN 0925-9724. doi: http://dx.doi.org/10.1007/BF00133655.

[62] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996. ISBN 0131829572.

[63] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Softw. Eng.*, 21 (4): 314–335, 1995. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.385970.

[64] I. C. Society. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, September 2000.

[65] C. Solís and N. Ali. ShyWiki-A Spatial Hypertext Wiki. In *Proceedings of the 2008 international symposyum on Wikis*. WikiSym '08, ACM, 2008.

[66] M.-A. D. Storey, D. Cubranic, and D. M. German. Ón the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, New York, NY, USA, 2005. ACM. ISBN 1-59593-073-6. doi: http://doi.acm.org/10.1145/1056018.1056045.

[67] The Open Group. *ArchiMate 1.0 Specification*. VAN HAREN PUBLISHING, May 2009.

[68] J. B. Tran, M. W. Godfrey, E. H. Lee, and R. C. Holt. Architectural repair of open source software. *International Conference on Program Comprehension*, 0: 48, 2000. ISSN 1092-8138. doi: http://doi.ieeecomputersociety.org/10.1109/WPC.2000.852479.

[69] H. Unphon. Making Use of Architecture throughout the Software Life Cycle—How the Build Hierarchy can Facilitate Product Line Development. Vancouver, Canada, May 2009. The Forth Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009).

[70] H. Unphon and Y. Dittrich. Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture. pages 178–183, Innsbruck, Austria, 2008. IASTED International Conference on Software Engineering.

[71] H. Unphon, M. A. Babar, and Y. Dittrich. Identifying and Understanding Software Architecture Evaluation Practices. Technical report (in progress), 2009. work in progress.

[72] H. Unphon, Y. Dittrich, and A. Hubaux. Taking Care of Cooperation when Evolving Socially Embedded Systems: The PloneMeeting Case. Vancouver, Canada, May 2009. The Cooperative and Human Aspects of Software Engineering 2009 (CHASE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009).

[73] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag Berlin Heidelberg, 2007.

[74] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, February 2008.

ARCHITECTURE-LEVEL EVOLVABILITY ASSESSMENT

Hataichanok Unphon
IT University of Copenhagen & DHI Water Environment Health
Denmark
unphon@itu.dk

ABSTRACT. This paper proposes a method for Architecture-Level Evolvability Assessment (ALEA). The method has been developed and implemented during a cooperative project with a company developing product line architecture for hydraulic modelling software. ALEA aims to evaluate how well the current design architecture can accommodate future use and business contexts. ALEA not only broadens prospects of architectural changes, but also takes sustainability of the changes into account. To assess sustainability, ALEA applies an evolvability framework consisting of sufficient contexts to propagate the effects of architectural changes. Based on a case study, empirical evidence of validating ALEA and the evolvability framework are presented. Confirmed by practitioners, ALEA offers a good structure for architecture assessment. Furthermore, practitioners favour its comprehensiveness, illustrations, and visualisation of the evolvability framework.

1. Introduction

Discovering a method to enhance software evolvability from re-engineering and modularising software products prompts our research work. This work aims to analyse how well the current design architecture aligns with business and use contexts. By looking at the current design architecture, the stakeholders are able to assess possible architectural changes and the effects of the changes on architectural contexts.

The main contribution of this paper is to propose Architecture-Level Evolvability Assessment (ALEA). ALEA has been developed, implemented, and validated during a cooperative project with a company developing product line architecture for surface water modelling systems. Due to the environment, user practices, and business vision, the architecture needs to allow for intensive tailoring and continuous development. ALEA provides the necessary elements for analysing design architecture. A framework proposed in [26] successfully complements a keystone of ALEA from the socio-technical perspective. However, there are some challenges to ALEA which should be further refined in order to support the evaluation more pragmatically. To avoid terminological confusion among architecture analysis, architecture assessment, architecture evaluation and architecture review, they are used interchangeably in this paper.

This paper is outlined as follows. Section 2 presents the case description. Section 3 explains our research method. Section 4 introduces terms and definitions. Section 5 reviews architecture evaluations. Section 6 elaborates on ALEA. Section 7 shows the implementation and evaluation of ALEA on the case study. Section 8 is discussion. Section 9 draws conclusions and looks at future works.

2. Case description

DHI Water Environment Health (DHI) is a pioneering organisation that develops software applications for hydraulic modelling [1]. In 1972, System 11 and System 21 were two of the first computational modelling systems developed at DHI to simulate water flow patterns with the help of one-dimensional and two-dimensional models. A three-dimensional simulation was developed in the 1980s. Originally, the organisation focused on hydraulic research, not on software engineering. Software development and software maintenance were challenges only on a small scale. All simulation programs were built in a similar way, i.e., an engine implementing differential equations changes the data in a set up model for one time step per simulation loop. In the late 1980s, DHI released the MIKE 11 and the MOUSE software products. Both products originated from System 11 following the requests of different usages, i.e. open channels and pipe networks. MIKE 11 and MOUSE are standalone Windows-based applications. The main users of these products are consultants who do simulations of hydraulic conditions, i.e. water level and flow, and analyse the hydrological effects of environmental change. Due to different market needs, ownership was split into different consultancy departments and in the last decades MIKE 11 and MOUSE have been developed and maintained in parallel. Released in 2005, MIKE URBAN followed requests to have a more complete and integrated modelling framework for both water supply and wastewater systems.

Through decades of successful use and development, the requirements of the software have evolved as well. In particular, the software is used in a more general setting, e.g. scheduled forecasts. The company was faced with the challenge of identifying and developing a kernel for data handling, simulation setup, and graphical interaction with simulations and their results. The first re-engineering project started with the MIKE 11 engine in 2006. Later, the MOUSE engine was merged into the MIKE 11 re-engineering project. Meanwhile, the organisation was changing. DHI set up a software product department in order to strengthen the software development process and the design. The software product department has taken development activities and ownership of DHI's software products. As a consequence, the department decided to re-engineer the core computational parts of some of the one-dimensional simulation software products, i.e. MIKE 11, MOUSE and MIKE URBAN, in a project called MIKE 1D. The project is estimated for 360 person weeks of implementation.

Lately, the software product department officially promoted another project called the Decision Support System (DSS) Platform. The DSS Platform affords end users the leverage to customise ongoing water simulation using historical, current, and predictive data. The DSS Platform usually uses data that has already been gathered into persistent storage and occasionally works from operational data. The simulation it builds on has to be set up as well by developing the model of the water system.

3. Research method

The research cooperation with DHI addressed the introduction of product line architecture into product development. The basis for the research described here is the fieldwork which I have been involved in for two and a half years. I wrote a research diary documenting daily observations, interviews, and meetings. As a field worker, I was expected not only to observe, but also to influence the projects in which I participated. The research was designed as action research by following the cooperative method development approach (CMD) [12]. The research activities are summarised in Table 1. Due to a lengthy period of cooperation, research activities are chronologically divided into three cycles: 1.) MIKE 11 re-engineering project, 2.) merging of MIKE 11 and MOUSE re-engineering project, and 3.) MIKE 1D project. Note that the research activities in the second cycle were collected when the third cycle was under way. Each cycle consists of three phases, i.e., participant observation, deliberating change, and evaluation. Most empirical evidence presented in this paper is obtained from the last cycle.

4. Terms and definitions

This section introduces (1) evolvability of socially embedded systems and (2) evolvability framework, as shown in Subsections 4.1 and 4.2 respectively. To avoid terminological confusions between system and software, the two terms are used interchangeably.

4.1. Evolvability of socially embedded systems

Unphon et al. [26] have defined *socially embedded systems* as *any system that can be modelled intensively according to the environment and practices of its end users*. ERP systems, e-government applications, virtual office software, and decision support systems are examples of socially embedded systems. Design decisions of socially embedded systems underline the importance of human interaction. According to Lehman [21], an *E*mbedded program (*E*-program) is a part of the world which it models. This implies a constant pressure for change. The usability of the system is the main concern of *E*-programs. Close cooperation between end users, people working with the systems on a daily basis, and developers throughout the entire development process is strongly recommended for capturing the contexts and qualities of use that cannot be fully anticipated at the initial phase. In use-oriented design, Participatory Design (PD) is regarded as a method

Table 1. Summary of research activities

Phase \ Cycle	**1.) MIKE 11 re-engineering project (Aug. – Nov.06)**	**2.) Merging of MIKE 11 and MOUSE engines re-engineering project (Dec.06 – Oct.07)**	**3.) MIKE 1D project (Feb.07 – Mar.09)**
Participant observation	- Study functionalities and code architecture of MIKE 11 and MOUSE engines. - Compare between MIKE 11 and MOUSE engine source code. - Interview DHI staff members. - Found a striking similarity in the source code between MIKE 11 and MOUSE engines.	- Review of architectural documentation and online user references systems used at DHI. - Observe development practices and technical infrastructure of MIKE 11 and MOUSE engines. - Review off-the-shelf documentation generators. - Interview developers and internal users of MIKE 11 and MOUSE engines on how they can use the architecture document.	- Review off-the-shelf static code analysis tools. - Analyse MIKE 1D source code using the reviewed tools and identify the relative complexity of its components. - Compare the analysis with the previous cycle projects. - Join MIKE 1D project weekly meetings. - Interview MIKE 1D team members on the idea of assessing the architecture and how they can use of the architecture as an aspect of software development.
Deliberating change	- Present a poster highlighting identical code parts between MIKE 11 and MOUSE engines. - Present a talk on software architecture and product line architecture. - Participate in a subproject on developing data access module architecture for the MIKE 11 re-engineering project.	- Propose a layered architecture to represent architectural knowledge. - Compare documentation generators and recommend a suitable one. - Update architecture documentation. - Create a prototype of an online architectural knowledge system.	- Conduct a workshop on architecture discovery with MIKE 1D team members. - Introduce the basic idea of architectural conformity checking. - Recommend suitable static code analysis tools. - Present the “good” and “bad” parts of the source code from the static code analysis tools. - Present an empirical study on architecture evaluation in industrial practice, the concept of software evolvability, and evolvability framework. - Propose Architecture-Level Evolvability Assessment (ALEA). - Organise a workshop on MIKE 1D and DSS compatibility.
Evaluation	- Evaluate the flexibility of the data access module by looking at different change scenarios at DHI and their implications in terms of implementation efforts. - Found that organisation of software development influenced product line architecture development [25].	- Found that architectural knowledge was more visible in the discussion than in the document. - Found that the prototype of the online architectural knowledge system has been set up and used internally.	- Found that architectural analysis tools and techniques embedded in daily routine were welcome by the development team. - Found that the development team uses “build hierarchy” to check the compliance of their source code against the architecture’s structure when they build the software [**Error! Reference source not found.**]. - Validate ALEA and evolvability framework with MIKE 1D team members.

for improving usability [18].

Socially embedded systems often allow users to tailor the software to specific needs. Examples of end user tailoring categories are customisation, composition, expansion, and extension [13]. Apart from tailoring, socially embedded systems must also evolve over time. Belady and Lehman [6] first introduced and used the term *evolution* as 'a sequence of changes to the system over its lifetime which encompasses both development and maintenance'. In today's competitive software market, it would be too restrictive to limit evolvability to maintenance issues only. The growth dynamics of a system depend highly on the business context. To increase market share, it may be vital to bring out new features. Yet, a system that is used will be changed [20]. Unphon et al. [26] have further defined *evolvability* as *the adaptability of software in order to serve the needs of use and business contexts over time reflecting on its architecture*. Architecture represents a common abstraction of a system that many of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus, and communication [5]. Architecture and other contexts around it must be adapted to accommodate the needs of use and business contexts.

4.2. Evolvability framework

We apply the evolvability framework proposed in [26] to review the effects of architectural changes as shown in Figure 1. The framework presents interaction between architecture and the six contextual dimensions, i.e., business, use, software engineering organisation, software engineering practice, technical infrastructure, and technical selection. Each contextual dimension is defined and illustrated as follows:

Business context is the context or environment to which the system belongs. For example, DHI software is a commercial software product and sold as licensed.

Use context relates the system to the work practices of the intended users. For example, hydraulic engineers use DHI software for water flow modelling, wave simulation, or flood forecasting.

Software engineering organisation is the organisational context in which the software development is carried out. For example, DHI software is developed in Denmark, the Czech Republic, and China.

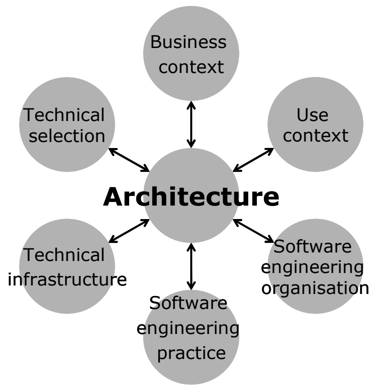


Figure 1. Evolvability framework

DHI software product department employs Microsoft Solutions Framework (MSF) team model [2]. MIKE 11 and MIKE URBAN software products were developed by different departments.

Software engineering practice refers to the analysis of the work practices of the system developers. For example, the development process at DHI is a mixture between iterative/incremental processes and agile methods. The core computational simulation developers are educated in hydraulic engineering, but the graphic user interface (GUI) developers are computer scientists. Most if not all MIKE 1D developers are highly educated in water and environmental engineering, not software engineering.

Technical infrastructure lists the hardware and basic software assets backing the system, focusing on the design as it is now. For example, MIKE 1D components are implemented in the C# programming language. The MIKE 1D project has unit test, nightly build, and build hierarchy as development infrastructure. DHI software only supports the Microsoft operating system.

Technical selection is part of a suggested design and relevant to design implementation. It needs to be seen in the context of existing and planned systems, as well as in the context of other systems that are part of the same design. For example, a common data access module handles setup data of MIKE 11 and MOUSE.

Others have used the notion of context or contextual factors before. Kensing [17] proposed a conceptual framework that IT designers should be aware of when they design applications for a specific organisation. The framework addresses: (1) project context, separating into design and implementation; (2) use context, dealing with work practice and strategy; and (3) technical context, interacting with system and platform contexts. Kensing does not apply the framework to concrete design proposals. Dittrich and Lindeberg [11] developed Kensing's framework further by mapping out contextual factors in order to understand the suitability of a less technically advanced design for a specific industrial setting. We further develop this framework to support architecture-based analysis when planning to evolve software products.

5. Architecture evaluations

This section reviews architecture evaluations from state-of-the-art and industrial practice, as shown in Subsections 5.1 and 5.2 respectively.

5.1. State-of-the-art

Ali et al. [4] classified and compared eight software architecture evaluation methods: Scenario-based Architecture Analysis Method (SAAM) [15], Architecture Trade-off Analysis Method (ATAM) [14], Active Reviews for Intermediate Design (ARID) [10], SAAM for Evolution and Reusability (SAAMER) [22], Architecture-Level Modifiability Analysis (ALMA) [8], Scenario-Based Architecture Re-engineering (SBAR) [7], SAAM for Complex Scenario (SAAMCS) [19], and Integrating SAAM in domain-Centric and Reuse-based development (ISAAMCR) [24]. Kazman et al. [16] contended that the categorisation and comparison are based on features of the methods instead of effectiveness and usability of the methods. The effectiveness of a method refers to producing results of real benefit to the stakeholders in a predictable, repeatable way. The usability of a method refers to capability of understanding and executing by its participants, learning reasonably quickly, and performing cost effectively. They further suggested four fundamental criteria for analysing an architecture evaluation method as follows: (1) context and goal identification, (2) focus and properties under examination, (3) analysis support, and (4) determining analysis outcomes. Nonetheless, both works point out features and criteria that need to be addressed when creating an architecture evaluation method.

5.2. Industrial practice

Unphon et al. [27] reported an empirical study on architecture evaluation in industrial practice. Findings of the study were based on interviews of ten architects who evaluated architecture of their organisations and other organisations. The findings showed a diversity of processes, participants and evaluation criteria in how architecture evaluation can be done in different organisations or business domains. Moreover, the findings also revealed preparation and challenges of architecture evaluation. The findings stated that non-technical issues (process, people, organisation, communication, and finance) can be the root cause of various architectural problems. Consequently, non-technical issues were also seen as the challenges of architecture evaluation. The findings confirmed that interaction between business and technical stakeholders is significant throughout the architecture evaluation. They emphasised the crucial role of an architect, i.e. (1) a middleman between business requirements and technical development, and (2) a responsible person for intellectual control over software being developed. Although the findings might not be generalised to any organisation, they provide good advice when designing a method to evaluate architecture.

6. Architecture-Level Evolvability Assessment

Architecture-Level Evolvability Assessment (ALEA) is a method to analyse how well the architecture supports future use and business contexts. The main concern of ALEA is sustainability of architectural change. Meaning that, if the current architecture is changed, how will the envisioned architecture look? The effects of changes will be checked with respect to quality factors and evolvability framework.

ALEA has been designed for assessing architecture of socially embedded systems. ALEA participants are stakeholders of the systems, e.g. end users, developers, a chief architect, and the project manager. To identify an assessment item, ALEA promotes interaction between business and technical stakeholders. Instead of assuming change or predicting use, each assessment item comes from stakeholders who tell what is expected to happen. It is vital that a “walking architecture” is involved throughout the assessment. The walking architecture is a chief

1. **Elicitation**
- Elicit existing architecture
- Elicit quality factors
- Identify an assessment goal
- Identify and prioritise assessment items

2. **Assessment**

   For each assessment item
- Architecture adaptation
  - Assess the existing architecture with respect to assessment item
  - Envision the architecture
  - Assess the envisioned architecture with respect to relevant quality factors
- Sustainability assessment
  - Assess the envisioned architecture with respect to evolvability framework

3. **Reporting**
- Document the whole assessment
- Follow-up

Figure 2. Summary of ALEA

architect or a main developer who carries most if not all the architectural knowledge and makes design decisions. ALEA entails a mechanism of follow-up by producing assessment complete assessment report. The details of ALEA methods and report templates are shown in Subsections 6.1 and 6.2.

6.1. ALEA method

A summary of ALEA method is shown in Figure 2. The ALEA method is divided into 3 stages: *elicitation*, *assessment* and *reporting*. The *elicitation* stage aims to prepare necessary elements for the assessment stage. The elements are *existing architecture*, *quality factors*, an *assessment goal* and *assessment items*. The *existing architecture* can be elicited from an architecture document or a "walking architecture". The *quality factors* [23] represent behavioural characteristics of a system. They include correctness, reliability, flexibility, testability, maintainability and reusability. The *assessment goal* is a purpose of the assessment. If the goal is not specifically identified, it will lead to involving unnecessary stakeholders and difficulties in identifying assessment items. The *assessment items* can be seen as new requirements, use scenarios [9], change issues, etc. If assessment items are abundantly identified, they must be prioritised. The assessment will start from the high-priority items.

Architecture-Level Evolvability Assessment

PART I

Goal: [Description]$^{+}$
Quality factors: [List of quality attributes]$^{+}$
Existing Architecture: [Figure and description]$^{+}$
Asessment items: [List of assessment items]+

PART II

Assessment item: [Prioritised number, assessment item name, description]$^{+}$
Architectureal discussion: [Description]$^{+}$
Envisioned architecture: [Figure and description]*
Related quality factors: [Description]*
Sustainability discussion
□ **Business context:** [Description]*
□ **Use context:** [Description]*
□ **Software engineering organisation:** [Description]*
□ **Software engineering practice:** [Description]*
□ **Technical infrastructure:** [Description]*
□ **Technical selection:** [Description]*
Conclusion: [Description]$^{+}$
Action plan: [Description]*

Note [ ]$^{+}$ is a required field. [ ]* is an optional field.

Figure 3. ALEA report template

The second stage is *assessment*. This can be seen as a loop in which *architecture adaptation* and *sustainability* of the adaptation will be reviewed for each item. *Architecture adaptation* includes (1) assessing existing architecture with respect to assessment item, (2) envisioning architecture, and (3) assessing envisioned architecture with respect to relevant quality factors. The *envisioned architecture* can be seen as the existing architecture with new components added, adding new interfaces to existing components, changing existing components, and changing existing interfaces. However, the design decision for the envisioned architecture will support or impede quality factors. On the other hand, the analysis with respect to quality factors provide a sound basis for making an objective decision in case of design trade-offs.

Sustainability assessment addresses the envisioned architecture with respect to the evolvability framework. If need be, some contexts might be adapted to support the envisioned architecture or the assessment item, or the envisioned architecture has to be refined. Because the envisioned architecture will "inhabit" the same context as the existing architecture, it is vital to be aware of what the root context of an assessment item is, which contexts could be affected, and how they could be adapted. In case the envisioned architecture has to be refined, the quality factors will be assessed again.

The *reporting* stage aims to *document* the whole assessment, which entails a mechanism of *follow-up*. It is absolutely essential that all findings are backed by evidence. For example, a problem in the architecture should come from stakeholder input. The mechanism of *follow-up* makes the design decision visible to responsible stakeholders. The mechanism is not to take decision immediately, but to inform and broaden solutions. For example, if the stakeholders are aware of what they gain from a possible solution, will they favour it or will they find another solution? There can be multiple solutions for the same assessment item, so the stakeholders can see which quality factors or evolvability contexts are affected by each solution.

6.2. ALEA report template

The ALEA method needs to produce a complete document for the assessment. Figure 3 shows an ALEA report template which is consistent with the ALEA method. The template has two parts, Part I and Part II, corresponding to the elicitation and assessment stages. Every field in Part I is required. Part II has both required and optional fields.

Part I captures a goal, quality factors, existing architecture, and assessment items. Part II captures a set of assessment items along with their architecture discussion, envisioned architecture, related quality factors, sustainability discussion, conclusion, and action plan.

7. Implementation and evaluation of ALEA at DHI

This section presents empirical evidence in which we implemented and evaluated architecture-level evolvability assessment (ALEA) in a case study. Subsection 7.1 describes the implementation of the ALEA at DHI. Subsection 7.2 presented lessons learned.

7.1. MIKE 1D and DSS Platform compatibility

When MIKE 1D team members and I implemented the ALEA method at DHI, our ambition was to ensure that the MIKE 1D ongoing development would align with the DHI business vision. One of the focuses at the MIKE 1D project is to support the uses of DSS Platform. We then prompted a workshop on MIKE 1D and DSS Platform compatibility. Our aim was to assess whether MIKE 1D architecture is good enough for DSS Platform usage. The workshop had two parts. The first part aimed at eliciting assessment items from the DSS Platform project. The second part aimed at discussing the items based on the ALEA method.

Due to limited funding, we arranged the first part of the workshop as a lunch meeting. Participants were not only team members of the MIKE 1D project and the DSS Platform project, but also all the interested stakeholders from DHI consultancy departments. The participants got information on current MIKE 1D design architecture, and ideas on how the DSS Platform could work with the MIKE 1D architecture. The participants gave direct input to the MIKE 1D team.

The input or assessment items were discussed in the second part of the workshop. Participants were MIKE 1D team members, an architecture expert and me. We discussed assessment items based on the ALEA method. At the end of the workshop, we reflected on the ALEA method and evolvability framework. After the workshop, the discussion was documented following the ALEA report template.

The MIKE 1D design architecture is shown in Figure 4. It consists of four layers: *Application*, *Controller*, *Data access*, and *Utilities*. Each layer comprises a number of components. Each component has its own interface which can be accessed by the other components. Only the *Data access* layer has two sub-layers of grouping components. Arrows show the “uses” relationship of components, layers, and products. For example, *Application MIKE URBAN* component uses *MU Proxy* components.

The *Application* layer contains *Application* and *Application MIKE URBAN* that can be run from a command prompt. The *Controller* layer handles the simulation of a water model. The *Controller* layer contains *MIKE 1D Engine*, *MIKE 1D Controller*, and *MIKE 1D Engine*

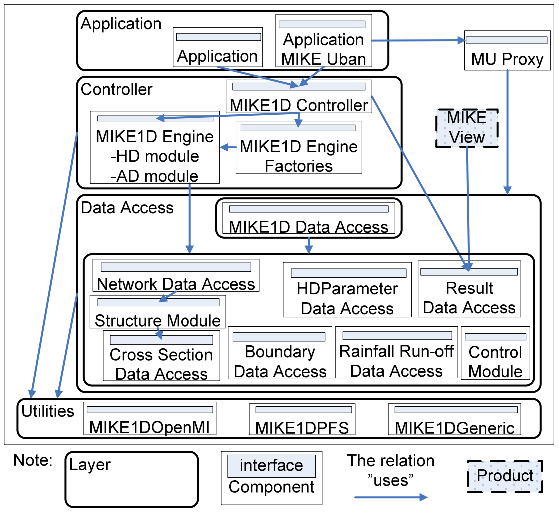


Figure 4. MIKE 1D design architecture

Factories. The *Data Access* layer handles the setup of the water model. The *Data Access* layer contains the *MIKE 1D Data Access* component and the other self-contained components, e.g. *Network Data Access*, *HD Parameter Data Access*, and *Boundary Data Access* components. The *Utilities* layer provides generic components which can be used by the higher layer components.

Design decisions of MIKE 1D architecture promote quality factors, such as maintainability, usability and integrity. For example, through the *Data Access* layer, the *MIKE 1D Engine* component, GUIs, and third-party users can handle the setup of a water model in a straightforward manner without accessing any persistent storage directly. As long as they can handle these data access components, they can set up or simulate a water model. The *MIKE 1D Data Access* component can read data from MIKE URBAN and MIKE 11 files and populates the setup data. Potentially, if a user wants to simulate a specific water model from a specific file, the user creates a specific file reader for that file and populates the *MIKE 1D Data Access* component. Then the user can perform simulation without changing anything in the *MIKE 1D Engine* component. Also, the user can validate the data before performing the simulation.

In the workshop, a member of the DSS Platform project posed an assessment item regarding setup data manipulation, which can be seen as a new requirement. He works with an operational flow forecasting system. One functional requirement of the DSS Platform is to handle "what if" situations. End users are free to change setup data like water inflow during a simulation. To forecast the next simulation, the DSS platform needs to know which setup data to use, results from previous simulation, manual input or calibrated setup data, e.g. when starting the forecast after a computing failure.

MIKE 1D team members suggested writing a "wrapper" around the *Data Access* layer. The wrapper is a minimal interface component that gives high-level functionalities to the *Data Access* layer. The wrapper would get data from the previous simulation, the calibrated setup data, or another persistent storage, e.g. a database or a result file from another simulation system. Thus, the wrapper would require metadata for transforming data appropriately. To create such a wrapper, nothing would change in the existing MIKE 1D design architecture. The architecture already supported the necessary extraction of the metadata. But an envisioned architecture would add the wrapper beside the *Data Access* layer, which can be done outside the MIKE 1D architecture.

In sustainability discussion, the MIKE 1D team members saw that the manipulation of setup data originated from the context of DSS Platform use. The envisioned architecture coined good questions on software engineering organisation and business context, i.e. "*With the current organisational structure, who should implement the wrapper? The MIKE 1D team, the DSS Platform team, or someone else?*" and "*Will the wrapper be one of DHI's saleable components? If so, who will take the lead on that?*".

To follow up the manipulation of setup data, the MIKE 1D team created the extra component on top of the existing MIKE 1D design architecture. They were aware that they should collaborate with the DSS Platform team to achieve the goal, i.e. MIKE 1D and DSS Platform compatibility.

7.2. Lessons learned

As reported in literature [25], organisation issues influenced the architecture. The funding model resulted in identifying an assessment goal and planning for the workshop. The first evidence is identification of assessment goal, i.e. MIKE 1D and DSS Platform compatibility. Comparing to another potential assessment goal, this goal can be assessed (a) between two in-house projects, and (b) without hiring any external hydraulic and environmental consultants. The second evidence is the workshop schedule. The workshop was split into two parts: elicitation and assessment. Elicitation was scheduled as a lunch meeting. Assessment was scheduled as an internal meeting. At DHI, the lunch meeting is considered as an internal meeting in which the host shall not spend extra budget for any participants because it is considered as part of common contribution. Thus, holding such a meeting means economics collaboration between different in-house projects or departments.

At the end of the workshop, the MIKE 1D team members gave feedback on the architecture-level evolvability assessment (ALEA). The team members found that the term “evolvability” is a rather abstract and difficult concept in itself. Some of the MIKE 1D team members wondered why it was discussed in their project. Eliciting quality factors was another challenge. The team members were not familiar with the term “quality factors”. Thus, giving a clear definition and showing examples could help better understand the ALEA method on the first implementation.

Elicitation of the assessment item is crucial. During the workshop, a DSS Platform team member raised a known issue which the participants discussed. When the MIKE 1D team had internal architectural discussion, one of its team members complained that “*I don’t know how they do that in practice actually.*”

The MIKE 1D team members suggested improving this method by either giving extremely precise instructions or eliciting a practical assessment item. The later suggestion can be seen as participatory design. More specifically, DSS Platform team members should have come up with how they actually handle the concrete online system. This could be done by showing how the DSS Platform team members use MIKE 11 or MIKE URBAN and generalising the concrete case.

The summary could help MIKE 1D team members to frame an overall idea of an assessment item and to connect it to actual practice.

After experiencing ALEA, the MIKE 1D team members approved of the structure, transparent decision-making process and trade-off analysis for product-line architecture. Before the method was introduced to the MIKE 1D team members, they assessed the architecture informally at the whiteboard. One of the members reported that "*When we do it (architecture assessment on the whiteboard), I think we get only half of the quality factors and half of the contexts (of the evolvability framework) because it is not structured. By getting this structure, we are able to make a more sound decision about what to do.*" Apart from that, the ALEA endorsed product-line architecture. One difficulty at DHI was thinking in terms of product line architecture. Often, a developer just came up with an idea to solve a problem. Due to this pragmatic decision, the developer often added his solution directly into the source code without considering whether it could be used for future projects. With the ALEA, the developer is encouraged to consider the consequences of change not only on his own project, but also sustainability in relation to other projects.

After the first implementation of the method, the members gradually learn the terms used in the method and connections between the architecture and its relevant contexts. At the time of writing, the members have planned to assess their architecture at the beginning of each milestone. "*It would be good tool for a project leader.*", one of the members suggested.

8. Discussion

This section presents notes on evolvability framework, and how ALEA can fit in a FramewOrk for Comparing Software Architecture Analysis Methods (FOCSAAM) [3] and industrial practice, as shown in Subsections 8.1 and 8.2.

8.1. Evolvability framework: comprehension and stakeholder affiliation

Positive feedback by MIKE 1D team members on evolvability framework are comprehensiveness, illustrations, and visualisation. The evolvability framework, as shown in Figure 1, was used in

Table 2. Summary of ALEA based on FOCSAAM

FOCSAAM		ALEA
Component	**Elements**	**Brief explanation**
Context	Software architecture definition	Structure(s) of system which comprise software elements, the externally visible properties of those elements, and the relationships among them [5]
	Specific goal	Change impact analysis
	Quality attributes	Evolvability and other elicited quality factors
	Applicable stage	All stages of software life cycle
	Input & Output	Embedded in method description
	Application domain	Socially embedded systems
Stakeholders	Benefits	Continuous quality check and specific benefit according to the assessment goal
	Involved Stakeholders	“Walking architecture” and selected stakeholders depending on the assessment item
	Process support	Embedded in method description, participatory design (recommended)
	Socio-technical issues	Embedded in method description
	Required resources	Funding, person hours spent for elicitation, assessment and reporting
Contents	Method’s activities	Three main stages: elicitation, assessment and reporting
	Software architecture description	Design architecture and code architecture
	Evaluation approaches	Based on change requirements, an expert evaluation
	Tool support	Evolvability framework
Reliability	Maturity of method	Developing and continuous validation
	Method’s validation	Case study

sustainability discussion. I found that the team members can visualise the consequence and propagate the effects of an envisioned architecture in a short period of time. What impressed me the most was the accuracy with which team members were able to predict the consequences suggested changes to the architecture.

When the evolvability framework was introduced, a team member questioned (1) the difference between technical selection and technical infrastructure and (2) how the framework relates to stakeholders. The answer to the first question is defined and illustrated in the Subsection 4.2. The answer to the second question is that the stakeholders can belong to contextual dimensions. For example, based on an assessment item of MIKE 1D and DSS Platform compatibility, as mentioned in Subsection 7.1, the DSS Platform team represents the use context.

8.2. How ALEA can fit into FOCSAAM and industrial practice

Ali Babar and Kitchenham [3] have developed a FramewOrk for Comparing Software Architecture Analysis Methods (FOCSAAM). We apply FOCSAAM to assess ALEA, as shown in Table 2. We found that ALEA provides essential features, as does most of the well-established architecture evaluation methods presented in [4]. ALEA contained evolvability framework as its tool support in the sustainability discussion section. Note that evolvability is the central "quality attribute" of ALEA. But the "quality factors" are elicited using ALEA. We consider effects on the quality factors as a part of evolvability. For example, in the continuous development of a system, the quality factors of the new version can differ significantly from those of the old version.

ALEA uses face-to-face conversation to convey information rather than documentation. However, ALEA reports are used to complement follow-up mechanisms. The follow-up mechanism is basically face-to-face conversation to "the right people" or responsible stakeholders. Based on an assessment item of MIKE 1D and DSS Platform compatibility, as mentioned in Subsection 7.1, MIKE 1D team members and DSS Platform team members are potential candidates for developing a wrapper. Due to the software engineering organisation at DHI, this decision will be taken by a head of the development group.

Validated by DHI case study, ALEA looks promising in terms of effectiveness and usability. Main reasons could be (a) the concept of architecture was "concretised" in the development environment before ALEA was implemented; (b) ALEA is designed based on industrial practice, especially DHI context; and (c) ALEA gives precedence to socio-technical perspective.

9. Conclusions and future works

This paper proposes a method called Architecture-Level Evolvability Assessment (ALEA) to evaluate how well the current design architecture can accommodate future use and business context. ALEA has been developed, implemented, and validated during a cooperative project with DHI Water Environment and Health. After the first validation, we found that Participatory Design (PD) can complement the elicitation stage of ALEA. Due to ALEA having been designed for review architecture of socially embedded systems, engaging end users beyond the design phase is unavoidable. It is somewhat misleading to state that ALEA is complete in its existing form. Rather, we expect ALEA to be refined with a wide range of assessment goals and items in order to better support architecture evaluation. Our ambitions for the refined ALEA are (a) comprehensible for a novice, and (b) applicable to continuous development of software product lines. Future research will have to show whether and how ALEA can be applied in different contexts.

References

[1] DHI Water Environment Health. [Online]. Available: http://www.dhigroup.com

[2] MSF Team Model v.3.1, Microsoft Solutions Framework (MSF) Team Model. [Online]. Available: http://www.microsoft.com/downloads/details.aspx?familyid=C54114A3-7CC6-4FA7-AB09-2083C768E9AB&displaylang=en

[3] M. Ali Babar and B. Kitchenham, "Assessment of a Framework for Comparing Software Architecture Analysis Methods," in *Proceedings 11th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, B. Kitchenham, P. Brereton, and M. Turner, Eds., Keele University, UK, 2 - 3 April 2007. [Online]. Available: http://www.bcs.org/upload/pdf/ewic_ea07_paper2.pdf

[4] M. Ali Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," in *Proceedings Australian Software Engineering Conference (ASWEC)*, 2004, pp. 309–318.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[6] L. Belady and M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 1, pp. 225–252, 1976.

[7] P. Bengtsson and J. Bosch, "Scenario-based software architecture reengineering," *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pp. 308–317, Jun 1998.

[8] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (alma)," *J. Syst. Softw.*, vol. 69, no. 1-2, pp. 129–147, 2004.

[9] J. M. Carroll, *Making Use: Scenario-Based Design of Human-Computer Interactions*, 1st ed. The MIT Press, 2000.

[10] P. C. Clements, "Active Reviews for Intermediate Designs," SEI, Carnegie Mellon University, Tech. Rep. CMU/SEI-2000-TN-009, 2000.

[11] Y. Dittrich and O. Lindeberg, "Designing for changing work and business practices," in *Adaptive evolutionary information systems*. USA: IGI Publishing, 2003, pp. 152–171.

[12] Y. Dittrich, K. Rönkkö, J. Eriksson, C. Hansson, and O. Lindeberg, "Cooperative method development," *Empirical Software Engineering*, vol. 13, no. 3, pp. 231–260, 2008.

[13] J. Eriksson, "Supporting the Cooperative Design Process of End-User Tailoring," Ph.D. dissertation, Department of Interaction and System Design, School of Engineering, Blekinge Institute of Technology, Sweden, 2008.

[14] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method," in *Proceedings of*

the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Monterey, CA, August 1998, pp. 68–78.

[15] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," in *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 81–90.

[16] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. Northrop, "A Basis for Analyzing Software Architecture Analysis Methods," *Software Quality Control*, vol. 13, no. 4, pp. 329–355, 2005.

[17] F. Kensing, "Participatory Design in a Commercial Context - a conceptual framework." New York, USA: Participatory Design Conference, 2000.

[18] F. Kensing and J. Blomberg, "Participatory Design: Issues and Concerns," *Computer Supported Cooperative Work (CSCW)*, vol. 7, no. 3-4, pp. 167–185, September 1998.

[19] N. Lassing, D. Rijsenbrij, and H. van Vliet, "On Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isn't Everything," in *Proc. Second Nordic Software Architecture Workshop (NOSA '99)*, 1999, pp. 1103–1581.

[20] M. Lehman, "On Understanding Law, Evolution, and Conservation in the Large-Program Life Cycle," *Systems and Software*, vol. 1, no. 3, pp. 213–231, 1980.

[21] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.

[22] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An Approach to Software Architecture Analysis for Evolution and Reusability," in *Proceedings of CASCON '97*, Toronto, ON, November 1997.

[23] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality," U.S. Department of Commerce, Washington, DC, Tech. Rep. RADC-TR-77-369, 1977.

[24] G. Molter, "Integrating SAAM in Domain-Centric and Reuse-based Development Processes," in *Proc. of the 2nd Nordic Workshop on Software Architecture (NOSA)*, 1999.

[25] H. Unphon and Y. Dittrich, "Organisation matters: How the Organisation of Software Development Influences the Development of Product Line Architecture." Innsbruck, Austria: IASTED International Conference on Software Engineering, 2008, pp. 178–183.

[26] H. Unphon, Y. Dittrich, and A. Hubaux, "Taking Care of Cooperation when Evolving Socially Embedded Systems: The PloneMeeting Case." Vancouver, Canada: The Cooperative and Human Aspects of Software Engineering 2009 (CHASE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), May 2009.

[27] H. Unphon, M. A. Babar, and Y. Dittrich, “Identifying and Understanding Software Architecture Evaluation Practices,” Technical report (in progress), 2009.

INTRODUCING AN EVOLVABLE PRODUCT LINE ARCHITECTURE

Hataichanok Unphon

IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark;
and DHI Water Environment Health,
Agern Allé 5, DK-2970 Hørsholm, Denmark

e-mail: unphon@itu.dk

Direct phone: +45 7218 5358
Mobile: +45 6072 9789
Fax: +45 7218 5001
www.itu.dk/people/unphon

ABSTRACT. The main contribution of this work is to develop an engineering discipline for maintaining software evolvability that is suitable for industrial practice. This paper presents a case study on: (*i*) how software architecture and product line architecture (PLA) has been introduced into the development practice, and (*ii*) how the architecture can be maintained over time. During two and one-half years of research collaboration with DHI Water Environment Health (DHI), we implemented (*i*) a conceptual framework providing consistent terminology for architectural changes, (*ii*) a work practice promoting adoptability and sustainability of architectural changes, and (*iii*) a supportive technical work practice for architectural changes through agile software development. This work combines conceptual, human, and technical levels in such a way that the development of the PLA does not jeopardise the continuous software evolvability.

KEYWORDS

Product line architecture • Architecture evaluation • Qualitative empirical research • Software engineering organisation

1. Introduction

Software evolution (Belady and Lehman 1976) can be described in different ways: e.g., software aging (Parnas 1994), maintainability (Lehman 1980; Bennett 1996; Cook et al. 2000), or refactoring (Mens and Tourwe 2004). But software also evolves when brand-new features, different development paradigms, or shifting business and organisational goals are introduced (Breivold 2008). Moreover, the software that is used will continue to evolve (Lehman 1980a). Software evolution becomes much more interdisciplinary, combining technical and non-technical aspects, with increasing concern for end users (Bennett and Rajlich 2000). Software that is required for change must be designed in such a way that the architecture is capable of meeting current needs, while providing a means of accommodating likely change throughout the software lifespan (Rowe et al. 1998). However, architecture evolution has typically been done in an ad-hoc manner (Chaki et al. 2009). The motivation of this work is to develop an engineering discipline for maintaining evolvable architecture that is suited for industrial practice.

This work was conducted as a research project with DHI Water Environment Health (DHI)[1]. DHI is an independent, international consulting and research organisation specialising in the area of water, environment and health. DHI offers a wide range of consulting services and leading-edge technologies, software tools, chemical and biological laboratories, and physical model test facilities, as well as field surveys and monitoring programmes. The research project initially addressed the re-engineering of one of DHI's software products, and introduced thinking in terms of product line architecture (PLA). After two and one-half years, PLA thoughts had begun to crystallise, especially after

[1] DHI Water Environment Health website: http://www.dhigroup.com

the architecture has been employed beyond the design phase of product line development. This work confirmed that organisation of software development influenced the development of PLA. Furthermore, an evolvability framework, a build hierarchy[2] and an Architecture-Level Evolvability Assessment (ALEA) have been proposed for guidance in mediating evolution throughout the whole software life cycle. The proposed engineering discipline reveals that the dynamic behaviour of software evolution, referred to as "evolvability" in Unphon et al. (2009), does not only cover the adaptability, but also the sustainability of continuous development and use. Furthermore, the proposed engineering discipline ensures that the development of the PLA does not jeopardise the continuous evolvability.

This paper is outlined as follows. Section 2 introduces the case description and the research approach. Section 3 presents related research. Section 4 elaborates on the introduction of PLA at DHI. Section 5 is discussion. Section 6 concludes the paper and outlines future work.

2. Case description and research approach

2.1. DHI Water Environment Health (DHI) case

DHI Water Environment Health (DHI) is a pioneering organisation that develops software applications for hydraulic modelling. In 1972, System 11 and System 21 were two of the first computational modelling systems developed at DHI to simulate water flow patterns with the help of one-dimensional and two-dimensional models. A three-dimensional simulation was developed in the 1980s. Originally, the organisation focused on hydraulic characteristics research, not on software engineering. Software development and software maintenance were challenges only on a small scale.

In the late 1980s, DHI released the MIKE 11 and the MOUSE software products. Both products originated from System 11 following requests for different usages, i.e., open channels and pipe networks.

[2] A build hierarchy is a technique to generate executable code based on dependencies between components.

MIKE 11 and MOUSE are stand-alone Windows-based applications. The main users of these products are hydraulic and environmental consultants who perform simulations of hydraulic conditions (e.g., water level and flow), and analyse the hydrological effects of environmental change. Due to varying market needs, ownership was split into different consultancy departments; and during the past three decades MIKE 11 and MOUSE have been developed and maintained in parallel. Released in 2005, MIKE URBAN followed requests to have a more complete and integrated modelling framework for both water supply and wastewater systems.

After decades of successful use and development, the requirements of the software have evolved as well. In particular there is a growing tendency that the software be used in a more general setting, e.g., scripts and scheduled forecasts. The company was faced with the challenge of identifying and developing a kernel for data handling, simulation setup, and graphical interaction with simulations and their results. The first re-engineering project started with the MIKE 11 engine in 2006. Later on, the MOUSE engine was merged into the MIKE 11 re-engineering project. The existing source code for MIKE 11 and MOUSE totals approximately 550,000 lines.

Meanwhile, the organisation was changing. DHI set up a software product department in order to strengthen the software development process and its design. The software product department has taken charge of development activities and ownership of DHI's software products. As a consequence, the department decided to re-engineer the core computational parts of some of the one-dimensional simulation software products – MIKE 11, MOUSE and MIKE URBAN – into a project called MIKE 1D. The project is estimated to require 360 person-weeks merely for implementation.

2.2. Research approach

The motivation of this research is to deliver a framework, a tool, and an evaluation method for engineering architecture evolution that is suitable for industrial practice. This research was designed as action research by following cooperative method development (CMD) (Dittrich et al. 2008). CMD is a domain-specific adaptation of action research consisting of three evolutional phases: 1) understanding practice; 2) deliberating improvement; and 3) implementing and

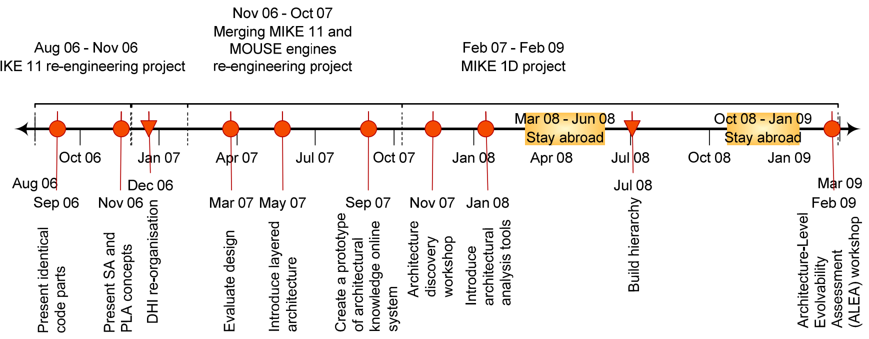


Figure 1. Research activities timeline

observing improvements. The research analysis described here is based on a research diary documenting daily observations, talks, and weekly meetings. In addition, transcripts of recordings of formal interviews and workshops are also analysed. Research activities are chronologically divided into three cycles: 1) the MIKE 11 re-engineering project; 2) the merging of MIKE 11 and MOUSE re-engineering project; and 3) the MIKE 1D project. Figure 1 shows the timeline of the three cycles, together with prominent research activities. Note that the research activities in the second cycle were collected when the third cycle was underway. The main reason for the overlap is that the third cycle officially began after the number of MIKE 1D project members had increased and become established. During the two and one-half years of research collaboration, research activities were temporarily suspended twice when I, as a fieldworker, travelled abroad. At the end of my fieldwork study, the research project members' feedback and the research results were discussed with MIKE 1D team members and external specialists.

3. Related research

In Bosch (2000), a software product line is described as consisting of a common architecture for a set of related products – also known as product line architecture (PLA) – and a set of reusable components that are designed for incorporation into the architecture. In addition, the software product line consists of software products that are developed using the aforementioned reusable assets. Architecture represents a common abstraction of a system that many of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus, and communication (Bass et al. 2003). The stakeholders refer to people or organisations who will be affected by the system and who have a direct or indirect influence on the system requirements (Kotonya and Sommerville 1998). Software architectural design is one of the key elements in the software product line. The design reflects how well an organisation can field software products that are built efficiently from the reusable assets. On the other hand, organisation of software development also reflects the design (Ganesan et al. 2006), as discussed in Section 4. Much has been written about software product line case studies (Bosch 2000; Clements and Northrop 2001; Pohl et al. 2005). But the DHI case study is unlike the others, with the following noteworthy exceptions.

First, DHI software development practice follows agile software development (Cockburn 2001), inspired by eXtreme Programming (XP) (Beck 1999) and test-driven development (Beck 2002). In XP, planning, analysing, and designing are done a little at a time. Test-driven development is related to the test-first programming concepts of XP. Software product line engineering starts with a domain engineering phase (Pohl et al. 2005). Domain engineering requires a substantial amount of work in terms of up-front analysis, which goes against the core principles and beliefs of XP (Ghanam and Maurer 2009).

Second, the design decisions for PLA at DHI include the preparation for adaptations via end user development. DHI software products are extensively tailored by end users. Some of the end users are allowed to integrate adaptation of the code into the mainstream development (Subsection 4.1). Although PLA addresses some of the design issues, the design is resolved within the development organisation. There is no systematic categorisation of different possibilities for end user development (Dittrich 2007). Apart from that, most of the product line frameworks originate from "technically embedded systems," where design decisions are constrained by interfaces with hardware or mechanical specifications. But this work is constrained by use contexts that cannot be fully anticipated at the initial phase.

Third, PLA research rarely mentions code architecture conformance checking (Bischofberger et al. 2004), and vice versa. During the whole software lifecycle, in particular the programming, a planned solution at the start of the project usually signals that design architecture will control code architecture. The design architecture can be thought of as the ideal implementation structure. The code architecture describes how the source code, binaries, and libraries are organised in the development environment (Soni et al. 1995) and implement the design architecture (Sub-section 4.3). When the project is in progress, the code architecture might, for a good reason, not conform to the design architecture. For instance, the code architecture may reveal an infeasibility of the planned solution. Thus, the design architecture must be adjusted in order to align itself with the code architecture. The problems, then, are (*i*) how to describe the compliance between the design architecture and the code architecture, and (*ii*) how to maintain it throughout the development. These problems are among the most important issues addressed in this work.

Fourth, PLA research focuses on adaptability (Böckle et al. 2002) or modifiability (Bengtsson et al. 2004), but this work takes sustainability into account (Section 4.3). The challenge when introducing PLA is how to assure the stakeholders that the PLA will support future uses and business contexts. If need be, the PLA is adapted to serve the needs of use and business contexts. However, the adaptation often affects or jeopardises the other contexts for continuous evolution. In order to prevent such a situation, sustainability is taken into account in engineering discipline.

Finally, different things become relevant. Carrying out PLA is more intricate than the architecture of a one-of-a-kind product. The main reasons are (*i*) most software assets rely on a set of products and versions, and (*ii*) multiple organisational units are involved (Bosch 2000; Unphon 2009b).

One contribution of this work is to increase the body of knowledge on the implementation of PLA in a development environment in such a way that the development of PLA does not jeopardise its continuing evolvability. Furthermore, this work addresses the role of architecture on the continuous evolution of a product line. Compliance between design architecture and code architecture is important and must be maintained throughout the software life cycle.

4. Introducing product line architecture at DHI

This section reveals how PLA has been introduced and implemented at DHI into three sub-sections. Each sub-section represents its own cycle, as mentioned in Section 3. Sub-section 4.1 presents the MIKE 11 re-engineering project. Sub-section 4.2 shows the merging of MIKE 11 and MOUSE engines re-engineering project. Sub-section 4.3 describes the MIKE 1D project. Each sub-section explains research activities, architectural practice changes, effects on software engineering organisation and practice, effects on other contextual dimensions, and research outcomes. Note that the outline of each sub-section is rather unusual for the sake of readability. The explanation of the research activities follows the three evolutionary phases of CMD. Only the first cycle did not include the final CMD phase because of the shift to the second cycle.

4.1. The first cycle: the MIKE 11 re-engineering project

The first cycle began in August 2006 as an initial fieldwork study with the MIKE 11 re-engineering project. The cycle ended when the project was officially expanded in November 2006. In this cycle, the members were a project leader and two developers, all educated in hydraulic engineering.

4.1.1. Research Activities

Understanding practices. Initially, my participation began by observing the re-engineering and modularising of the existing MIKE 11 product. I studied the code architecture and functionality of the MIKE 11 product, as well as its corresponding products, i.e., MOUSE. I compared the similarity of the source code between the MIKE 11 and MOUSE engines. I found some GUI functionality that I believed should have been done in an *engine* or a separate module. The engine is the main computational part. All the functionalities were poorly organized, and on the same file and the same unit.

Most of the DHI software products' end users have a BSc or MSc in hydraulic engineering, and are able to program on their own; many of them used DHI's applications during their studies. In an organisational context, I found that each consultancy department had its own software development team and consultancy team. Coordination between two teams in the same department was easily achieved. The consultancy team fed its development of new hydraulic modelling elements into the mainstream development branch of the respective department. But the growth of the organisation was not optimised because of double implementation of software development: for example, assigning developers to solve the same task, thus spending unnecessary resources on development and maintenance.

Deliberating improvements. After finding a striking similarity in the source code between MIKE 11 and MOUSE, I presented a poster highlighting the identical code parts in order to initiate a discussion among the software developers. Subsequently, I made a presentation on software architecture and PLA. Later I participated in a subproject that discussed new data access module architecture, and developed a prototype of the module. In the meantime, I conducted informal interviews with DHI staff members.

4.1.2. Architectural practice changes

The benefit of merging the MIKE 11 and MOUSE engines was clearly visible, and project members became well aware of that benefit. MIKE 11 and MOUSE have their own data access patterns. MIKE 11 defines data structures in an XML-like proprietary format, and stores the data in a file. But MOUSE and MIKE URBAN use a database. When the project members designed a new data access module, they took these data access patterns into consideration. Project members, however, struggled with understanding MOUSE architecture from source code and data format standpoints. Spontaneous discussions on the architecture of the data access module took place, with various conceptions drawn on a white board.

4.1.3. Effects on software engineering organisation and practice

The MIKE 11 and MOUSE engines were developed in parallel, as they were derived from different departments. Initially, the re-engineering project was conducted by one department. The company's internal policy for collaboration among departments is time-consuming and costly. Thus, coordination with another department seemed impossible. When both departments had a meeting on merging the re-engineering project, one of the departments had questions about resource allocation and budget approval for the joint project. After the meeting, I conducted separate interviews with members of the two departments, during which the concept of *"we"* and *"they"* became obvious. One team member suggested that collaboration would be possible once the output of the re-engineering project had been delivered. Another member was eager to join, and was willing to reduce the gap in development between the two departments. In December 2006 the internal re-organisation of the software development and consultancy postponed the approval of the budget for both departments, and rescheduled a merging of the MIKE 11 and MOUSE engine re-engineering project.

4.1.4. Effects on other contextual dimensions

Most of the hydraulic engineers at DHI were familiar with the C, C++, Delphi, or Fortran programming languages. Therefore, applications were implemented in these languages. In order to support safer dynamic memory allocation, technology benchmarking, communication with other components in the .NET environment, and continuity in recruiting new developers, it was decided to change the programming language for implementation of the data access module from Delphi to either Fortran, unmanaged C++, managed C++, or C#. After some discussion and evaluation of the pros and cons, in the context of DHI, a decision to use the programming language C# was made.

Due to a lack of explicit architectural documentation, the coordination between the two departments in terms of suggesting or implementing substantial software changes was difficult. When both departments had a meeting on joining the re-engineering project, the MIKE 11 developers presented the prototype of a common part by showing C# source code. The prototype was discussed with the MOUSE developers.

4.1.5. Research outcomes

When MIKE 11 and MOUSE evolved from System 11, their explicit architectural representations were not fully documented. Only a "walking architecture" was available. Walking architecture is a term for

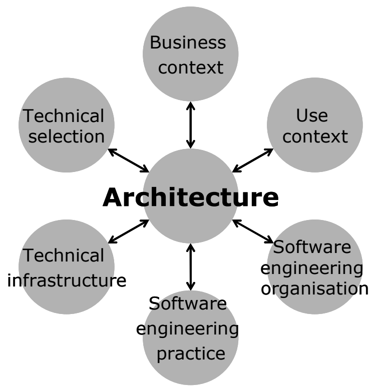


Figure 2. Evolvability framework

the chief architect or main developer who possesses most, if not all, of the architectural knowledge and, consequently, makes crucial design decisions. When a developer has to implement some part of the code without understanding 100% of the existing code, he would usually copy and paste from the old code. However, the assumption for this practice was a lack of architectural documentation. If the architecture documentation exits, does it solve the root of the problem(s)? This question brought us to the next cycle.

The first cycle showed that not only technical perspectives but also the organisation of software development and use issues influenced MIKE 11's architecture evolution (Unphon and Dittrich 2008). A framework comprised of six contextual dimensions around architecture – business context, use context, software engineering organisation, software engineering practice, technical infrastructure, and technical selection – was gradually developed. Figure 2 visualises the framework. Each contextual dimension is defined as follows:

Business context is the context or environment to which the system belongs.

Use context relates the system to the work practices of the intended users.

Software engineering organisation is the organisational context in which the software development is carried out.

Software engineering practice refers to the analysis of the work practices of the system developers.

Technical infrastructure lists the hardware and basic software assets backing the system, focusing on the design as it is now.

Technical selection is part of a suggested design and relevant to design implementation. It should be seen in the context of existing and planned systems, as well as in the context of other systems that are part of the same design.

4.2. The second cycle: the merging of MIKE 11 and MOUSE engines re-engineering project

This cycle started in November 2006 and ended in October 2007. In this cycle, a project leader was changed and a developer in the previous cycle resigned. Thus, the members now included a new project leader plus a developer. Both of them were educated in hydraulic engineering.

4.2.1. Research activities

Understanding practices. In order to be close to the project, I took it upon myself to create the architectural documentation. I had reviewed some of DHI's architectural documentation and online user references systems. I also observed the development practices and technical infrastructure of the MIKE 11 and MOUSE engines. I reviewed a number of documentation generators which automate technical document production from the source code. I interviewed developers and internal users of MIKE 11 and MOUSE about how they could make use of the architectural documentation.

Deliberating improvements. I organised an evaluation workshop with a group of DHI software specialists. A comparison was made between the traditional data access design in MIKE 11/MOUSE and a new design with a data access module. The flexibility of the new data access module was evaluated. The different change scenarios at DHI and their implications in terms of implementation efforts were inspected. Apart from that, I reported on a comparison of documentation generators and recommended a generator that was suitable for the project.

Implementing and observing improvements. After analysing the source code and understanding the practice of software development, I proposed layered architecture (Buschmann et al. 2007) to represent explicit design architecture. I created a prototype of an online architectural knowledge system. The system contained a project overview, architectural knowledge, user references, and examples. The overview of the project explained the project's vision. The architectural knowledge presented the overall design, layered architecture, and diagrams along with detailed explanations. The user references showed technical documentation, which was automatically generated from source code, e.g., class overviews, namespace overviews, and interface overviews. The examples described used scenarios of some components in six different programming languages: C#, Visual Basic (within Excel), MATLAB, Delphi, VBScript and JScript.

4.2.2. Architectural practice changes

During the DHI department re-organisation, the merging of the MIKE 11 and MOUSE engines re-engineering project proceeded slowly; but this was soon moved to the MIKE 1D project. The prototype of the online architectural knowledge system was set up and used internally. But architectural knowledge was still being discussed rather than documented. The role of "walking architecture" was recognised.

Since the architecture was explicitly defined, I found that the MIKE 1D's implementation followed design architecture. Figure 3 shows a sample of the initial design architecture, which represented a core

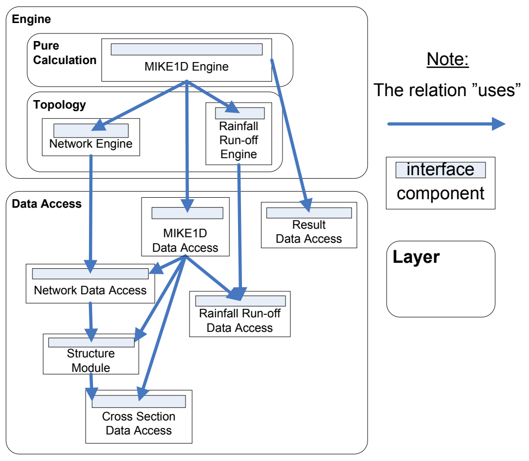


Figure 3. A sample of the initial design architecture

computational part of future MIKE 11 and MOUSE products. The entire core computational part was divided into a *Data Access* layer and an *Engine* layer. The *Engine* layer was further divided into two sub-layers: *Topology*[3] and *Pure Calculation*[4]. The *Data Access* layer was further divided into a number of components, as was the *Engine*. Later on, viable interfaces for these components were defined. The interfaces identified how the components should communicate with each other. The components were categorised into layers and assembled based on design rules, which defined acceptable dependencies between components: i.e., (a) prohibition of upward relationships – it is inherent in layered architecture that references from lower to upper layers are not allowed (in other words, only downward relationships are allowed); (b) interface violations – usage of non-interface artefacts of components by other components is not allowed; (c) several layer downward relationships are acceptable; and (d) prohibition of relationships within a layer – components in different lines of products should not relate to each other.

A considerable amount of component-based software engineering provides flexibility for handling changes at the level of design architecture. A component interface design helps developers introduce new functionalities with minimal impact on all changes. However, it is important that a "walking architecture" keeps an eye on any change that could possibly break the interface.

4.2.3. Effects on software engineering organisation and practice

Before the re-organisation, the development teams organised software to be in line with each department. After the re-organisation, DHI grouped software product development teams from different departments into one department. The DHI software product department was founded in December 2006, addressing a need to strengthen the software development process. Ownership of software products changed hands from consultancy departments to the software product department. This re-structuring of the organisation of software development accommodated the design of PLA.

[3] Topology handles static model data, e.g., network topology.

[4] Pure calculation handles dynamic model data, which is used in actual computations and the simulation state.

The new software product department employs the Microsoft Solutions Framework (MSF) team model[5]. The MSF team model is based on six interdependent, multidisciplinary roles: product management, program management, deployment, testing, user experience, and release management. The team roles are flexible enough to follow agile methods, especially in keeping documentation to a minimum. Release of software products changed from once a year to once every two years.

4.2.4. Effects on other contextual dimensions

In the evaluation workshop on the design of the data access module, one of the developers found that the new design did not allow for easy usage. The data access module was, he felt, set up in such a way that every developer had to learn five different languages in order to simply try something new. Another question was raised about the new design. What were the possibilities of experimenting freely with the software as it was? Meanwhile, DHI was implementing procedures to maintain collaboration and relationships between the development and consultancy departments without disrupting the work processes and development logistics required by the new organisational structure.

4.2.5. Research outcomes

In parallel with creating a prototype of an online architectural knowledge system, a project was begun to explicate design architecture. Project members began to discuss their software using software architecture terms: for instance, “layered architecture,” and “dependencies between modules.” However, the architecture was not concretised into development practice. The evolvability framework which had been developed in the first cycle, as shown in Figure 2, was still being used as a tool for structuring design discussions when drafting architecture and when evaluating architecture evolution. But the framework had not yet been introduced to any project members.

[5] MSF Team Model v.3.1, Microsoft Solutions Framework (MSF) Team Model. http://www.microsoft.com/downloads/details.aspx?familyid=C54114A3-7CC6-4FA7-AB09-2083C768E9AB&displaylang=en

4.3. The third cycle: the MIKE 1D project

The MIKE 1D project was announced in February 2007, but the cycle did not actually start until October 2007. The MIKE 1D project is still ongoing. But this cycle ended in March 2009, coinciding with the end of this fieldwork study. At the beginning of the cycle, a project leader was changed and two full-time developers and a part-time developer joined. Thus, the initial members of the team were a new project leader and four developers. Later on, a hydraulic consultant was brought in as a part-time software tester. Midway through this cycle the main developer resigned. Shortly after that, another full-time developer left the company. The company then recruited a new full-time developer. At the time of this writing, the members consist of a project manager, two full-time developers, a part-time developer, and a part-time tester. They are a mixed group with different educational backgrounds, i.e., hydraulics, mathematics and physics.

4.3.1. Research activities

Understanding practices. I reviewed static code analysis tools. Employing those tools, I analysed the MIKE 1D source code and identified a complexity measure (McCabe 1976) of the MIKE 1D components. I also compared the MIKE 1D project's source code with that of the MIKE 11 re-engineering project, and the merging of the MIKE 11 and MOUSE engines re-engineering project. I joined the weekly meetings of the MIKE 1D project and conducted informal interviews with MIKE 1D team members, e.g., an interview on how they made use of the architecture as an aspect of software development.

Deliberating improvements. Together with a group of software architecture experts, I conducted a workshop on architecture discovery with MIKE 1D team members. The basic idea of checking for architectural conformity was introduced to the team members. Tools for automated (*i*) checking source code and architecture at build time, (*ii*) continuous integration server, and (*iii*) checking source code for proper format were recommended. After finding the "good" and "bad" parts of the source code by using static code analysis tools, I presented my findings at the weekly meetings of the project. The source code

1. **Elicitation**
- Elicit existing architecture.
- Elicit quality factors.
- Identify an assessment goal.
- Identify and prioritise assessment items.

2. **Assessment**

   For each assessment item:
- Architecture adaptation
  - Assess the existing architecture with respect to assessment item.
  - Envision the architecture.
  - Assess the envisioned architecture with respect to relevant quality factors.
- Sustainability assessment
  - Assess the envisioned architecture with respect to evolvability framework: business context, use context, software engineering organisation, software engineering practice, technical infrastructure and technical selection.

3. **Reporting**
- Document the whole assessment.
- Follow-up.

Figure 4. Summary of the Architecture-Level Evolvability Assessment (ALEA) method

comparative analysis was presented in the form of a Kiviat graph[6]. The interdependencies between components were represented in a layered architecture and a dependency structure matrix.

Implementing and observing improvements. I presented an empirical study of architecture evaluation in industrial practice, including the concept of software evolvability, and an evolvability framework. I proposed the Architecture-Level Evolvability Assessment (ALEA) method to analyse how well the architecture would support future uses and business contexts. A summary of the ALEA method is shown in Figure 4. The main concern of ALEA is sustainability of architectural change: that if the current architecture were changed, how the envisioned architecture would work.

When the team members and I implemented ALEA, our ambition was to make sure that the ongoing MIKE 1D development would align with DHI's business vision. One of the focus areas of the MIKE 1D project was to support the use of a Decision Support System (DSS) platform. The DSS platform was another project promoted by the DHI software product department, which affords end users the leverage to customise ongoing hydraulic simulations using historical, current, and

[6] A Kiviat graph is a multi-vector line graph showing the interrelationship of multiple variables: for example, percentage of comment, number of methods per class, and average complexity.

predictive data. MIKE 1D team members and I organised a workshop on MIKE 1D and DSS platform compatibility following the ALEA method. The aim of the workshop was to assess whether MIKE 1D architecture was good enough to be used by the DSS platform.

4.3.2. Architectural practice changes

Part of the reluctance to work with an explicit architecture was the fear of having outdated documents and a diverging code base. Introducing a build hierarchy for architectural compliance checking in the daily routine was, therefore, welcomed by the development team. The build hierarchy was implied in the design architecture, so that components in upper layers must be built after those in a lower layer. For example, in Figure 3 the *Network Engine* component in the *Topology* layer must be built after the *Network Data Access* component, the *Structure Module* component, and the *Cross Section Data Access* component in the *Data Access* layer.

Currently, the build hierarchy is defined in such a way that developers specify the build order. But Microsoft Visual Studio has another way of handling the logic of a build hierarchy. Microsoft Visual Studio has a *solution*, a top collection of *projects*. MIKE 1D developers work under the same *solution*, i.e., the *MIKE 1D Solution*. In the solution, there is a list of projects. Each developer is responsible for his project(s) in the *MIKE 1D Solution*. Each project contains actual source code and its unit tests. Each project represents a component. Hence, the developers define the dependency between components through the projects. Afterwards, the developers can see in which order the projects are built. When the developers compile or build the solution, the build hierarchy will automatically check whether the developers have followed the design architecture. When the developers check out the source control system and re-compile or re-build the solution, they will be aware of what the other developers have been doing. The developers also use unit tests to assure that any functionality change will not break the architecture.

MIKE 1D developers iteratively work on the design architecture and continue to refine it. With the help of the build hierarchy, the developers are able to see if dependencies point in the wrong direction. In order to realign the order of dependency, the developers can tweak a number of other dependencies, or introduce a new component.

Consequently, the refined design architecture is reflected in the build hierarchy. Since the architecture has become modularised, components can be tested separately, and as each component is composed. The core components can also be replaced. For example, to change the equation of water flow in a core component, a developer can implement a specialised component with the same interface as that core component. Without knowing how the core component is internally implemented, the developer only sees the interface of the core component and implements his specialised functionality. Afterwards, he moves the core components out, and replaces the core components with the specialised component without impacting anything else (specifically in a DHI context) in the build hierarchy.

The architecture has been used intensively via the build order of Microsoft Visual Studio. All of the developers on the team have the MIKE 1D solution, and in that solution there is a list of projects. Each developer works on different projects; but they still use the same solution, and re-compile every ten minutes. And every ten minutes, they can also check out and re-build the solution and find out what the others have been doing. They follow eXtreme Programming (XP) practices (Beck 1999) and test-driven development (Beck 2002). Since they each have their own unit tests, they are immediately aware when someone checks in and updates the base module; and the higher-level module captures any changes using this unit testing. This solution works very well with all the unit tests. Thus, changes can be captured if the module compromises something else, or if it compromises someone who uses this module at a relatively early stage.

When software products are upgraded or released in a new version, changes at the asset base must comply with previously released versions. With the help of interface-based design, if developers want to change a particular component in the asset base, they can create a new component with the same interface as in the previous version. When the developers build software with the new component, the build hierarchy will notify developers whether the new component complies with the architecture of previous releases. Thus, maintainability – one of the MIKE 1D quality attributes – is influenced positively by the increased separation of components. However, the management of multiple organisational units should be optimised in parallel.

After becoming acquainted with ALEA, the MIKE 1D team members approved the structure, the transparent decision-making process, and the trade-off of analysis vis-à-vis PLA. Before the method was introduced to the MIKE 1D team members, they assessed the architecture informally at the whiteboard. One of the members reported that: "*When we do it (architecture assessment on the whiteboard), I think we get only half of the quality factors and half of the contexts (in the sustainability assessment) because it is not structured. By getting this structure, we are able to make a more sound decision about what to do.*"

4.3.3. Effects on software engineering organisation and practice

When new team members are introduced to the MIKE 1D project, their tasks are explained from an architectural point of view. The main developer illustrates the MIKE 1D design architecture, albeit not in much detail. But this helps the new members to begin working on the project. New members can easily picture how the components will fit together. This is a strong point of the MIKE 1D project, as one of the members said; the project has a walking architecture that essentially describes itself. New team members will initially be assigned to implement a self-contained component[7] of the design architecture. Thus, the new team members will not change any of the core components. New developers will use implementation templates to get started. When asked how to decide into which component he should put the physical equation, one of the developers who had been newly introduced to the team replied: "*Actually, the main developer showed me the component and told me to put the equation here and here. Then I started it. As for the framework of MIKE 1D, I didn't really know how it works….*" Another new member had experienced being a new developer in another project. At that time he had to attempt to understand the project strictly by deciphering the source code. "*That was a time-consuming task, but it's much more manageable now after I moved to MIKE 1D,*" he reported.

The idea of a build hierarchy is straightforward to the developers. MIKE 1D developers distribute the work after the architecture has been designed. They decide up front on a protocol for communication and dependencies between components. They can work on their own

[7] A self-contained component refers to an independent component or a component that is barely used by other components.

implementation without compromising each other's work. They hardly ever work on the same file. Even though their source control system improves, they do not need to merge a file as often as before because they are working on separate components. One of the MIKE 1D developers said, "*It's just way easier to handle it. It is much easier to test. It is just a lot easier for us to work with, and it works better.*" The idea of outsourcing MIKE 1D development to other developers who are familiar with both MOUSE and MIKE 11 engines was mentioned. The developers would thus be able to implement some of the components if they have time, or work on them in parallel without affecting other MIKE 1D developers. A milestone and release plan was decided upon from a functional point of view. However, thinking in terms of architecture and build hierarchy indirectly impacted on the plan, and introduced a new component.

MIKE 1D team members work in a common "airy" room where they can sit near each other. When it comes to architectural discussion, the "roommates" easily perceive "what and why" the architecture has been changed. As confirmed by the MIKE 1D team members, the open work space is one of the important elements that promotes decentralisation of architectural knowledge during development.

After the initial implementation of the ALEA method, the MIKE 1D team members gradually learned the terms used in the method, as well as the connections between the architecture and its relevant contexts. At the time of this writing, the members plan to assess their architecture at the beginning of each milestone. "*It would be a good tool for a project leader,*" one of the members suggested.

4.3.4. Effects on other contextual dimensions

Although the MIKE 1D project is still in the production phase (i.e., it is not yet finished at the time of writing), the developers are already beginning to see the benefits that MIKE 1D will eventually yield. Because of its flexibility in design, the users (i.e., hydraulic and environmental consultants) will be able to replace a particular component without impacting the whole software product. The users can tailor the core components by adding a specific component without changing any of the core components. Additionally, the users can change the non-interface code of a specific component without (a) waiting for the next release, or (b) having an impact on the general software product.

DHI is now willing to fully open the MIKE 1D data access interfaces and documentation for their customers and end users. The MIKE 1D team has already put some of the architecture on the online help page. MIKE 1D components will be used as core assets for the MIKE 11 product, as well as for MIKE URBAN or MOUSE products. End users will use the same product as before, except they will not have a product called MOUSE. They will use the MIKE URBAN product instead. Since the MIKE URBAN product uses the MOUSE product as one of its engines, and since the MOUSE engine has merged with the MIKE 11 engine to become the MIKE 1D engine, end users can use the MIKE URBAN product as they would the MOUSE product.

4.3.5. Research outcomes

During the entire software life cycle, architecture evolves. A build hierarchy becomes a key element in making architecture visible, and influencing day-to-day development practice. The build hierarchy facilitates code architecture conformance checking, and instantly reveals a divergent coincidence between the design architecture and the code architecture at the build time. Usages of the build hierarchy for product line development have been published in Unphon (2009b).

ALEA promotes thinking in terms of PLA. When a developer comes up with an idea to solve a problem, he often adds his solution directly into the source code without considering whether it could be used for future projects. With ALEA, the developer is encouraged to consider the consequences of change – not only on his own project, but also its sustainability in relation to other projects. The details of ALEA can be found in Unphon (2009a).

Based on interviews on the perspectives of using architecture (in workshops on the development of MIKE 1D and DSS compatibility) MIKE 1D team members raised interesting challenges regarding evolvability. These challenges confirmed that changes in the architecture or any of the contextual dimensions affect everything else.

Business context. One of the requirements for DSS platform usage is to handle “what if” situations in setup data manipulation. Based on the

MIKE 1D architecture, as shown in Figure 3, MIKE 1D team members suggested writing a "wrapper" around the *Data Access* layer. The wrapper is a minimal interface component that gives high-level functionalities to the *Data Access* layer. One of the questions to be considered is: "*Will the wrapper be one of DHI's saleable components?*"

Use context. The developers have designed MIKE 1D architecture in such a way that dependencies between components are clearly defined. Thus, users or hydraulic and environmental consultants can replace components with respect to the dependencies without any subsequent problems. But the dependencies do not guarantee that the users can model in a better way. If the developers are not aware of the user side, it will be more annoying for the users. For example, there was a discussion about how to specify hydraulic resistance. Previously, users had to specify the hydraulic resistance in many different files. Then, an architectural recommendation was made to do this in a specific way, in which the users would make more decisions and take time to set up their initial models. One of the users complained: "*The way I understand it, the data should be located together with __ data, and we could get it from that. You can do everything with it, but it will be terrible to work with the model in that way. You will have repeated the number over and over again. That is not really the way to do it. I do not want to copy my information. I want to have it all in one place. Oh, it should be 13, not 10! I do not want to change 2,000 places. I want to change one place.*" The truth of the matter is that the users would have slightly more work in some respects, but the architecture would reduce the users' work in other respects.

Technical infrastructure. MIKE 1D components are implemented in the C# programming language. But the software products predating MIKE 1D were implemented in the C++, C#, and Delphi programming languages. Handling the mixture of different languages would not be an easy task.

Technical selection. The ancestors of MIKE 1D had different approaches of implementing similar functionality. In fact, MIKE 1D's initial ambition was to create a common component between MIKE 11 and MOUSE. But there are some instances where the two approaches are equally good. For example, the time step calculations between MIKE 11 and MOUSE engines are different. MOUSE uses a smaller time interval, so the calculation takes longer than with MIKE 11. Due to the general difference in time scales used by applications of MOUSE and MIKE 11 engines, they are optimised differently with respect to stability and accuracy. These two different focuses should be maintained. Although the developers try to merge the two approaches of implementing the same functionality as much as possible, they sometimes end up in the situation that "we need both."

Software engineering organisation. The software product department develops general purpose software products which can be sold to customers in great numbers. The consultancy departments have implemented very specific functionalities based on the needs of their projects. It is usually not possible to include a specific functionality in generic products because the functionality is not practical for the common user. Thinking in terms of architecture is extremely important in the current DHI organisational structure. Without being aware of how the architecture looks, development in the consultancy department will diverge from that of the software product department. And if the consultancy department does not comply with the architecture in the software product department, they will potentially increase future maintenance, as they may not be able to re-use components across releases.

Software engineering practices. Everyone in the MIKE 1D project can program and gradually learn about the architecture. But it always ends up that only a "walking architecture" can set up the architecture for the others. One of the mentioned benefits is the interface-based design. However, at the same time, this design becomes one of the weaknesses. Many interfaces have to be maintained. If the developers change the interfaces frequently, it will be difficult for others to work on any components. Therefore, the challenge was and remains how to define a viable interface of the core components as early as possible.

5. Discussion

This section sums up the keys to success of this work; shows how the DHI case study complies with a framework for software product line practice; reviews related methods for architecture evaluation and software product line evaluation; and emphasises the importance of introducing and deliberating architecture concepts for communicating change (as shown in Sub-sections 5.1-5.4, respectively).

5.1. The keys to success: conceptual, human and technical levels

The main contribution of this work is to develop engineering discipline for maintaining evolvability in such a way that the development of PLA does not jeopardise the continuous evolution. The success of introducing PLA into product development at DHI covers three levels of concern: conceptual, human and technical. The conceptual level gives coherence to the state of the practice at the early states of this work (Sub-section 4.1), i.e., evolvability framework and walking architecture. The evolvability framework helps the researcher understand how design takes place. The walking architecture reflects how the architecture is presented in a DHI development environment. The human level refers to work practices and social interaction of stakeholders. The ALEA method and open workspace are considered to promote the importance of the human level, communication in particular. The ALEA method structures the discussion in such a way that project members take adaptability as well as sustainability into account when evaluating architectural changes from new requirements. Open workspace can be seen as informative workspace in XP primary practices. The technical level refers to a system or technical infrastructure, i.e., a build hierarchy and an online architectural knowledge system. The build hierarchy is an architecting technique for XP developers. The online architectural knowledge system combines architectural and technical documentation for developers and end users. At the time of this writing, the system has been used internally at DHI. Note that success at the technical level requires support at the human level. Synergy between the conceptual, human and technical levels is not only considered to be the key to success, but also will increase the architectural awareness of project members on the evolvable PLA development.

In Knodel et al. (2008), architecture compliance checking has been successfully applied to software product line engineering for technically embedded systems. Architecture compliance checking was performed by a tool that analysed and visualised the compliance between the design architecture and the code architecture. Later, compliance checking feedback was collected and discussed in workshops, after which decisions were made. However, a compliance checking cycle took a relatively long time; the shortest cycle lasted two months. Using a build hierarchy, compliance checking takes ten minute per cycle, i.e., at the build time. The shortened cycle is suitable for XP or any feedback-obsessive practice in agile methods. A comparison of agile methods and software product line approaches presented in Tian and Cooper (2006) gives an idea for tailoring the software product line method with agility, although they have a few conflicts (Hanssen and Fægri 2008) – in particular, design for changes (Carbon et al. 2006). However, this work has already suggested solutions to the conflicts between conceptual, human and technical levels, as mentioned as the keys to success. In addition, the suggested solutions are relevant to software product companies seeking ways to increase agility and productivity through agile methods and software product line engineering.

5.2. A framework for software product line practices

Among the number of exiting frameworks, a framework from the most cited book[8] of software product lines is selected to show how the DHI case study put development and management activities into an existing framework for a software product line. The Software Engineering Institute (SEI) has continuously developed a mature framework (Clements and Northrop 2001) that captures the latest information about successful software product line practices[9]. The framework proposes essential activities and practices supporting core asset development, product development and management. However, the framework does not give concrete instructions on implementing specific engineering tasks (Vehkomäki and Känsälä 2000). But a use for the framework in this paper is to explain product line planning and management at DHI, as follows.

5.2.1. Product line planning: core assets and product development

The MIKE 1D project can be seen as a core asset development, where the main requirement is to maintain the existing core

[8] Cited by 1,486 articles according to http://scholar.google.com (last visited 11 July 2009).

[9] A framework for software product line practice (version 5.0) is available online at http://www.sei.cmu.edu/productlines/framework.html

functionalities of the MIKE 11, MIKE URBAN, and MOUSE products. These basically simulate one-dimensional water flow models for rivers, urban water supplies and sewers, and rainstorm events. In other words, MIKE 11 and MIKE URBAN products use MIKE 1D core components. MIKE 1D builds up on unit systems, time series objects, and license systems, etc. DHI has a top-down architectural approach; components for a product line are pre-defined in a way that indicates what component should be included in the product using the license system. For example, it must be known which license the core components belong to. End users must have the appropriate license number, so that the core components which are used in the corresponding product will be executable. The DHI software sales department is responsible for controlling the version number and modules for the products. In the MIKE 1D project, they reused most of the algorithms or equations that were optimised in the MIKE 11 or MOUSE products. Since the programming language was changed from Delphi to C#, the MIKE 1D developers had to build the core assets from scratch. At the same time, they re-factored and restructured the code, and made an effort to write comments.

5.2.2. Management

Since MIKE 1D is used for existing products, maintenance and debugging costs can be reduced by following software product line engineering discipline. DHI has a reference group for the MIKE 1D project comprised of the heads of the consultancy departments, executive salespeople, a head of the customer care department and a head of the software department, as well as the MIKE 1D project manager. The group has regular meetings with regard to management of the project, and its risks and requirements. The group must ensure that the requirements will not disrupt the ideal architecture of MIKE 1D and corresponding products, while at the same time considering marketing aspects and making sure that the products satisfy end users. DHI has a customer care department and a sales department in charge of customer and supplier interface management. DHI also has a technology road map and release board. The technology road map is a plan with short-term and long-term goals, with recommended technological solutions to achieve those goals. The release board is a group of people who examine all products and vote for which to deliver at the forthcoming release and at future releases. DHI actively empowers itself as a product line champion. It is moving forward to develop other core assets for two- and three-dimensional water modelling and GUIs.

What goes beyond the framework are the two proposed testing techniques, build hierarchy and the ALEA method. The ALEA method is a kind of testing or validation of artefacts, i.e. the architecture. The build hierarchy and the ALEA method afford test-driven development the leverage to balance quality attributes usually presented in a product line.

5.3. Architecture evaluation and software product line evaluation

Since future uses and business contexts can change over time, architecture evaluation should be performed regularly to discover whether PLA is still evolvable, as proposed in the ALEA method. Contextual dimensions have been discussed extensively in other methods, such as Architecture-Level Modifiability Analysis (ALMA) (Bengtsson et al. 2004), Architecture Trade-off Analysis Method (ATAM) (Kazman et al. 2000), and Architecture Reviews for Intermediate Designs (ARID) (Clements 2000). However, the dimensions are not structured for analysis or review in these other methods. The main difference between ALEA and ALMA, the closest evaluation method, is that the ALEA method evaluates not only the modifiability but also the sustainability of the architecture with respect to an explicit evolvability framework. ALEA requires half a day, excluding preparation and preliminaries, instead of three full days spent on ATAM, or one to two days spent on ARID. That way architecture evaluation can be performed more frequently, and becomes part of the agile development cycle. Apart from that, the artefacts examined in the ALEA method cover both design architecture and code architecture.

Besides architecture evaluation, the evolvability framework is a decision-making tool to find a sustainable way to improve software product line engineering. A BAPO-based framework (Business-Architecture-Process-Organisation) introduced in the Family Evaluation Framework (FEF) (van der Linden et al. 2004) is a model similar to the evolvability framework. Mapping between the BAPO model and the evolvability framework can be seen as: B-Business context, A-Architecture, P-Software engineering practice, and O-

Software engineering organisation. The BAPO model identifies interrelationships among four independent software development concerns; applying changes in one concern induces changes in the other three concerns. Each has its own profile scale for benchmarking. However, an action that improves one of the profile scales may lead to a reduction in the values of the other scales. The BAPO model has been developed, mostly if not exclusively, from technically embedded systems. Therefore, the use context was not explicitly mentioned, as opposed to the evolvability framework. The evolvability framework was developed from socially embedded systems. Design decisions of socially embedded systems underline the importance of human interaction with (and cooperation via) the software in terms of societal activities. However our evolvability framework does not offer any scale for each contextual dimension, as explicitly stated in the BAPO model. But the evolvability framework introduced in the ALEA method actually helps stakeholders be aware of changes in one context that induce changes in the other contexts, and involving the right people. For instance, a requirement for MIKE 1D architecture was to support the future use by a DSS Platform. The envisioned MIKE 1D architecture that resulted from the architecture adaptation discussion was to have an extra component in addition to the current MIKE 1D architecture. In the sustainability discussion, MIKE 1D team members posed good questions regarding the software engineering organisation and business contexts: "*With the current organisational structure, who should implement the extra component? The MIKE 1D team member, the DSS Platform team, or someone else?*" and "*Will the extra component be one of DHI's saleable components? If so, who will take the lead on that?*" Furthermore, the MIKE 1D team members were aware that they should collaborate with the DSS Platform team to implement the requirements. This exemplifies how the ALEA method affords an organisation the leverage to adapt and sustain a software product line approach.

5.4. The importance of introducing and deliberating architecture concepts for communicating change

This work aims to take an academic approach and make it relevant to practitioners, while at the same time taking evidence from the practitioners and feeding both into the research to evolve established results. The most challenging aspect of this work is to bridge the gap between the terminology used in the academic and industrial worlds. Moving between one discipline and another, it is necessary to introduce a common "language" to talk about software architecture, software product lines, and software evolution. One of the MIKE 1D team members said: "*This [introducing architecture] is the success criteria for us as a company. When [a new team member] first came here, I spent half a day introducing him to the main concept of MIKE 1D architecture, and then he started to work.*" Another member confirmed that: "*On the first day, I looked at the code and saw it had been well designed. It is very easy to understand. That is the great thing.*" However, the introduced terminologies – e.g., evolvability, PLA, quality attribute, and stakeholder – are rather abstract for practitioners who are not academically trained in software engineering. One MIKE 1D developer complained that: "*I think it [terminology] has been one of the things limiting this project. We have difficulties understanding what we are talking about.*" Practitioners need concrete terminology that can provide answers to questions such as: "*What does evolvability mean in our context? What does PLA cover? Who are the stakeholders? And what are their interests? The stakeholders are always there, but why are they there?*"

An idea for improvement is to carefully introduce necessary terms, and use practitioners' language and discipline as a basis for communication. Having different languages to be able to talk in different ways about architecture also helps practitioners follow the terminology. Apart from that, the terminology should be re-introduced over time. When project members leave midway through a particular project, they take with them a great deal of embedded knowledge of what has already been introduced. Although the company can replace the vacant position, the embedded knowledge is lost. Another risk of introducing terminology is a terminology gap between people who are involved in the project and those who are not. For instance, when MIKE 1D project members showed MIKE 1D design architecture in a workshop on MIKE 1D and DSS Platform compatibility, the MIKE 1D project members were presenting modules, but the others talked about the API (application programming interface).

To summarise, the evolvability framework, the build hierarchy, and the ALEA method are proposed as engineering disciplines to introduce an evolvable PLA at DHI which would link up at conceptual, human, and technical levels. The ALEA method and the build hierarchy are seamlessly integrated into test-driven development's primary practices in such a way that the PLA can be regularly validated. As a consequence, awareness of the architecture is dramatically increased throughout the software life cycle. Compared to existing architecture evaluations and software product line evaluation, the evolvability framework and the ALEA method put more emphasis on the sustainability of architectural changes on a long-term basis.

6. Conclusions and future work

The DHI case study reveals how architecture evolution can be engineered. Re-engineering one-of-a-kind product development into product line development is one way to maintain the evolvability of DHI's software products. In particular, the architecture of the main computational part used in MIKE 11, MOUSE, and MIKE URBAN has evolved into MIKE 1D architecture. The engineering practices do not only concern the evolution of core assets, variations, and assembly of MIKE 11, MOUSE and MIKE URBAN, but also the adaptations by hydraulic and environmental engineers. Thus, the use of architecture is taken seriously throughout the process, not just during the design phase. This work proposes an evolvability framework, a build hierarchy, and an ALEA method to concretise architecture in the development environment and practice. This study shows that this concretisation has resulted in a number of improvements: software quality and flexibility; communication and cooperation among new team members; distribution of development tasks and parallel implementation; and foreseeable usage by hydraulic and environmental consultants. Empirical evidence from this case confirmed that developing PLA has been accommodated by re-structuring of the software engineering organisation.

Apart from that, terminology used in academia and industry should be further refined. Many software products are developed by non-computer scientists. Although they may be eager to follow the best software engineering discipline, they are sometimes obstructed by the daunting academic terminology. A simple solution would be to have different explanations in order to communicate effectively with both academia and industry. Practitioners have found that newly introduced terminology, tools, or methods are useless without incorporating them into daily practices in an industry context. Moreover, the terminology should be re-introduced over time, because the turnover of people in the industry is unavoidable.

At the time of this writing, MIKE 1D team members and I, as an academic researcher, are planning to write an internal report summarising this cooperative research project. We are simplifying the academic terminology for an industrial context, using concrete examples in describing the PLA at DHI. The report also recommends practices for other DHI software projects based on the findings of this research project.

References

Bass L, Clements P, Kazman R (2003) Software architecture in practice. 2nd ed. Addison-Wesley Professional, Don Mills, Ontario, Canada

Beck K (1999) Extreme programming explained: embrace change. Addison-Wesley Professional, Don Mills, Ontario, Canada

Beck K (2002) Test-driven development: by example. Addison-Wesley, Boston MA

Belady L, Lehman M (1976) A model of large program development. IBM System Journal, 15(1):225–252

Bengtsson P, Lassing N, Bosch J, van Vliet H (2004) Architecture-level modifiability analysis (ALMA). J Syst Softw, 69(1-2):129–147

Bennett K (1996) Software evolution: past, present and future. Information Software Technology, 38(11):673–680

Bennett KH, Rajlich VT (2000) Software maintenance and evolution: a roadmap. In: ICSE '00: Proceedings of the Conference on the Future of Software Engineering. New York NY, 73–87

Bischofberger W, Kühl J, Löffler S (2004) Sotograph - a pragmatic approach to source code architecture conformance checking. In: The 1st European Workshop on Software Architecture (EWSA 2004). St. Andrews, UK, 1–9

Böckle G, Muñoz JB, Knauber P, Krueger CW, do Prado Leite JCS, van der Linden F, Northrop L, Stark M, Weiss, DM (2002) Adopting and institutionalizing a product line culture. In: Proceedings of the 2nd International Conference of Software Product Lines. San Diego CA, 49–59

Bosch J (2000) Design and use of software architectures: adopting and evolving a product-line approach. Addison-Wesley, New York NY

Breivold HP Crnkovic I, Eriksson PJ (2008) Analyzing software evolvability. In: Computer Software and Applications, 2008 – 32nd Annual IEEE International. Turku, Finland, 327–330

Buschmann F, Henney K, Schmidt DC (2007) Pattern-oriented software architecture: a pattern language for distributed computing. Vol. 4. Wiley, Hoboken, NJ

Carbon R, Lindvall M, Muthig D, Costa P (2006) Integrating product line engineering and agile methods: flexible design up-front vs. incremental design. In: Proceedings of the 1st International

Workshop on Agile Product Line Engineering (APLE '06). Kyoto, Japan

Chaki S, Diaz-Pace A, Garlan D, Gurfinkel A, Ozkaya I (2009) Towards engineered architecture evolution. The Third Workshop on Modelling in Software Engineering (MiSE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009). Vancouver, Canada

Clements P, Northrop L (2001) Software product lines: practices and patterns. Addison-Wesley Longman, Boston MA

Clements PC (2000) Active reviews for intermediate designs. [Technical report CMU/SEI-2000-TN-009] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Cockburn A (2001) Agile software development. Addison-Wesley Professional, Don Mills, Ontario, Canada

Cook S, Ji H, Harrison R (2000) Software evolution and software evolvability. [Working paper] University of Reading, Reading, UK

Dittrich Y (2007) Rethinking the software life cycle: about the interlace of different design and development activities. In: Dagstuhl Seminar: End User Software Engineering. Dagstuhl, Germany

Dittrich Y, Rönkkö K, Eriksson J, Hansson C, Lindeberg O (2008) Cooperative method development. Empirical Software Engineering, 13(3):231–260

Ganesan D, Muthig D, Knodel J, Yoshimura K (2006) Discovering organisational aspects from the source code history log during the product line planning phase – a case study. In: Proceedings of the 13th Working Conference on Reverse Engineering. Benevento, Italy, 211–220

Ghanam Y, Maurer F (2009) Extreme product line engineering: managing variability & traceability via executable specifications. In: Agile Conference 2009. Chicago IL

Hanssen GK, Fægri TE (2008) Process fusion: An industrial case study on agile software product line engineering. Journal of Systems and Software, 81(6):843–854

Kazman R, Klein M, Clements P (2000) ATAM: Method for architecture evaluation. [Technical report CMU/SEI-2000-TR-004, ADA382629] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Knodel J, Muthig D, Haury U, Meier G (2008) Architecture compliance checking – experiences from successful technology transfer to industry. In: The 12th European Conference on Software Maintenance and Reengineering (CSMR 2008). Athens, Greece, 43–52

Kotonya G, Sommerville I (1998) Requirements engineering: processes and techniques. Wiley, Chichester, UK

Lehman M (1980a) On understanding law, evolution, and conservation in the large-program life cycle. Journal of Systems and Software, 1(3):213–231

Lehman M (1980b) Programs, life cycles, and laws of software evolution. Proceedings of IEEE special issue on Software Engineering, 68(9):1060–1076

McCabe TJ (1976) A complexity measure. In: ICSE '76: Proceedings of the 2nd international conference on software engineering. San Francisco CA, p 407

Mens T, Tourwe T (2004) A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2):126–139

Parnas DL (1994) Software aging. In: ICSE '94: Proceedings of the 16th international conference on software engineering. Sorrento, Italy, 279–287

Pohl K, Böckle G, van der Linden FJ (2005) Software product line engineering: foundations, principles and techniques. Springer, Berlin, Germany

Rowe D, Leaney J, Lowe D (1998) Defining systems evolvability - a taxonomy of change. In: IEEE International Conference and Workshop: Engineering of Computer-Based Systems. Jerusalem, Israel

Soni D, Nord RL, Hofmeister C (1995) Software architecture in industrial applications. In: Proceedings of the 17th International Conference on Software Engineering (ICSE'95). Seattle WA, 196–207

Tian K, Cooper K (2006) Agile and software product line methods: are they so different? In: Proceedings of the 1st International Workshop on Agile Product Line Engineering (APLE '06). Baltimore, MD

Unphon H (2009a) Architecture-level evolvability assessment. [Work in progress]

Unphon H (2009b) Making use of architecture throughout the software life cycle – how the build hierarchy can facilitate product line development. In: The Fourth Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009), in conjunction with the

2009 IEEE 31st International Conference on Software Engineering (ICSE 2009). Vancouver, Canada

Unphon H, Dittrich Y (2008) Organisation matters: How the organisation of software development influences the development of product line architecture. In: IASTED International Conference on Software Engineering. Innsbruck, Austria, 178–183

Unphon H, Dittrich Y, Hubaux A (2009) Taking care of cooperation when evolving socially embedded systems: the PloneMeeting case. In: The Cooperative and Human Aspects of Software Engineering 2009 (CHASE 2009), in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009). Vancouver, Canada

van der Linden F, Bosch J, Kamsties E, Känsälä K, Obbink H (2004) Software product family evaluation. Softw Product Lines, 3154:110–129

Vehkomäki T, Känsälä K (2000) A comparison of software product family process frameworks. In: IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families. Springer-Verlag, London, pp 135–145