Mälardalen University Press Dissertations
No. 32

SOFTWARE SYSTEMS IN-HOUSE INTEGRATION

OBSERVATIONS AND GUIDELINES CONCERNING ARCHITECTURE AND PROCESS

Rikard Land

2006

Department of Computer Science and Electronics
Mälardalen University

ISSN 1651-4238
ISBN 91-85485-20-9
Printed by Arkitektkopia, Västerås, Sweden

Abstract

Software evolution is a crucial activity for software organizations. A specific type of software evolution is the integration of previously isolated systems. The need for integration is often a consequence of different organizational changes, including merging of previously separate organizations. One goal of software integration is to increase the value to users of several systems by combining their functionality, another is to reduce functionality overlap. If the systems are completely owned and controlled in-house, there is an additional advantage in rationalizing the use of internal resources by decreasing the amount of software with essentially the same purpose. Despite in-house integration being common, this topic has received little attention from researchers. This thesis contributes to an increasing understanding of the problems associated with in-house integration and provides guidelines to the more efficient utilization of the existing systems and the personnel.

In the thesis, we combine two perspectives: software architecture and processes. The perspective of software architecture is used to show how compatibility analysis and development of integration alternatives can be performed rapidly at a high level of abstraction. The software process perspective has led to the identification of important characteristics and practices of the integration process. The guidelines provided in the thesis will help those performing future in-house integration to make well-founded decisions timely and efficiently.

The contributions are based on several integration projects in industry, which have been studied systematically in order to collect, evaluate and generalize their experiences.

Included Papers

This thesis includes six peer-reviewed research papers, published at international journals, conferences and workshops. The papers are introduced presented in section 2.4 (page 18), with my individual contribution clearly indicated, and reprinted in full (page 107 and forward).

Acknowledgements

There are many people I wish to thank for their part in this thesis coming into being during the past five years. To do so without yielding to sentimentality – which is not appropriate at all for an aspiring researcher – I will summarize these five years in objective numbers:

1 supervisor (thanks Ivica!),

1 tool implementation (thanks to Mathias Alexandersson, Sebastien Bourgeois, Marko Buražin, Mladen Čikara, Miroslav Lakotić, Lei Liu, and Marko Pecić),

2 participations in industrial projects, around two man-months each (thanks to the unnamed company and my colleagues there),

2 ½ children (thanks to Cecilia for delivering them, and thanks to Selma, Sofia and Kuckelimuck for giving my life some meaning that is not so easily caught in numbers),

6 months stay at FER (Faculty of Electrical Engineering and Computing) at the University of Zagreb (thanks to Prof. Mario Žagar and my room mates Damir Bartolin, Tomislav Čurin, Marin Orlić, also to Igor Čavrak for all collaboration),

20 published research papers and 4 technical reports (thanks to my fellow authors for pleasant cooperation: Laurens Blankers, Jan Carlson, Ivica Crnković, Igor Čavrak, Erik Gyllenswärd, Mladen Kap, Miroslav Lakotić, Johan Fredriksson, Stig Larsson, Peter Thilenius, Christina Wallin, Mario Žagar, Mikael Åkerholm),

23 formal interviews with people in industry, plus an unknown number of informal talks (thanks to all interviewees and their organizations),

31.5 years of experience from living (thanks to everyone involved in making life mostly a pleasant experience, and thanks also to whoever was involved in giving life to me in the first place – I am sure I did not deserve it),

1,500 (estimated number) cups of tea, coffee and kava sa šlagom at MdH/IDE (Department of Computer Science and Electronics at Mälardalen University) and FER (thanks to everyone who joined for nice chats, especially my colleagues at the Software Engineering Lab and some unnamed persons at the Computer Science Lab who were always in the coffee room before me),

11,000 (estimated number) kilometers on bike from my home to IDE (thanks to all who contributed to my new bike for my 30th birthday),

∞ love and support (thanks to Cecilia again, Selma, Sofia and the rest of my family)

Rikard Land, July 2006

Cover Art: SELMA LAND

Table of Contents

Chapter 1. Introduction

It is well known that successful software systems must be evolved to remain successful – as a consequence they are progressively modified in various ways and released anew [237,299,302]. A current trend is to increase the possibilities of integration and interoperability of software systems with others. This is achieved typically by supporting open or de facto standards [265] or (in the domain of enterprise information systems) through middleware [51]. This type of integration concerns information exchange between systems of mainly complementary functionality. There is, however, an important area of software system integration that so far, has been subject to little research, namely the integration of systems with overlapping functionality. For such overlapping systems, developed and controlled in-house (i.e. within a single organization), the problems involved in this kind of systems integration – although commonly occurring in practice – has been studied even less. I have (together with colleagues) labeled this type of integration *in-house integration*[1] for short (more precisely it should be labeled in-house integration of in-house controlled software systems[2]). There are several possible reasons for the gradual or sudden development of overlapping systems: the systems may initially have been built to address different problems in different parts of the organization but have evolved and expanded to include more and more functionality. Finally, the overlap is significant enough to attract the attention of management. Other, more

---

[1] As we use both the terms "integration" and "merge" in this thesis, let us clarify our usage briefly: *In-house Integration* describes the overall task of creating a new system given two or more existing, functionally overlapping, software systems within an organization. To achieve this, *Merge* is one strategy – among several – which means a tight integration.

[2] Existing systems developed and controlled in-house are often called "legacy systems". We have avoided this term, however, since it is often associated with characteristics in addition to merely being controlled in-house, such as being old and built with old technologies, having a degraded architecture, and being insufficiently documented, thus being difficult to understand and hard to change.

dramatic events include company acquisitions and mergers, and other types of close collaborations with other organizations. A new system combining the functionality of the existing systems would improve the situation in the sense of rationalizing internal resources, as well as from the points of view of users, customers, and other stakeholders.

An increasing number of products incorporate software, and there is also an increasing trend to the building and using software internally for use within a single organization. Reorganizations and company mergers are also common phenomena, which means that it is becoming increasingly important to be able to eliminate the overlap of software systems. Although many organizations have certainly encountered this challenge already, and more or less successfully handled it, their experiences have – to my knowledge – not been collected systematically across organizations and made publicly available.

In this thesis, I present a sequence of research studies collecting the experiences of organizations, analyzing these experiences and generalizing them into guidelines for future in-house integration projects.

1.1 Scope and Assumptions

I have viewed the problem of in-house integration mainly as a software engineering problem, and have chosen two complementary points of view from which to study the topic of in-house integration, namely *processes* and *software architecture*, motivated and described below:

- **Processes.** In-house integration is essentially a human endeavor, which can be seen as a set of activities in an organizational context. Important activities and stakeholders need to be identified – both at a high-level and in more concrete situations – so that decisions as well-founded as possible can be made rapidly, and so that the cost and time of the implementation process are predictable. If some important activities are omitted, the decisions may be ill-founded and the integration delayed and costly, or never completed, and/or the resulting integrated system may be of low quality.
- **Software Architecture.** The systems to be integrated are arguably among the most important artifacts to study and evaluate. They should be evaluated from a technical point of view as well as from the perspective of various stakeholders (users, managers, etc.). The need for early and rapid decisions has led me to focus on the architectures of the

systems, i.e. high-level descriptions of the systems. Many issues can and should be briefly discussed, in order to form a relatively high-level statement concerning important similarities and differences between the systems. In the thesis, the term software architecture means not only the well-known academic definitions concerning *structure* [25], but also other high-level design decisions with significant impact, in particular *data models* and *frameworks* used (in the sense "environment that defines components").

I am fully aware, however, that an organization must combine the knowledge and understanding of many other fields of research and practice to succeed with its in-house integration. Examples of other important issues to consider, outside the scope of this research, are how to properly handle the staff whose employment might depend on decisions concerning the future of existing systems, how to overcome cultural differences [150] and how to make the suggested processes and practices actually work [321]. Proper application of the theories and practices of management, business, and (organizational) psychology, would certainly contribute greatly to the success of an in-house integration project. This said, I believe that there are some pitfalls in the technical areas we have studied that may cause enormous inefficiencies or even failures if one fails to recognize them and manage them properly.

1.2 Research Questions

The question for an organization faced with the in-house integration challenge is *how to make decisions as good as possible, as rapidly as possible*. This thesis is intended to obtain an answer to this question.

Before proceeding, however, I would like to clarify several issues with this formulation. First, there is not an absolute optimum to be found in a mathematical sense of "as good/rapidly as possible". The answer to be expected is a set of suggested activities that should precede a decision; activities that can be carried out rapidly. Second, the "goodness" of a decision depends on perspective; in this thesis decisions and events are evaluated from the point of view of organizational economics, where a "good" decision would be one which allows an organization to make the transition efficiently (in terms of time and money) from the situation with functionally overlapping systems to a state with one single coherent system. (From other points of view the same decision could be considered disastrous, for example by the staff at a site that will be closed as a result of the

decision.) Third, even with this broad definition of a "good" decision, it is difficult or practically impossible to obtain an evaluation properly and unambiguously. From the economics point of view, one measure would be the overall turnover and profit of the organization. However, from a scientific standpoint one would need to know more particularly how much the integration contributed to this economic result, taking into account all direct and indirect effects. Also, one should expect integration to have a significant up-front cost, and it becomes problematic to define when it is most appropriate to evaluate the economic result.

I did not want to formulate a less interesting research question because it would be easier to answer, but, aware of these limitations, I set out to pursue the question I believe would give the most interesting answers, even if these answers can only be partial and incomplete. In line with the focus on process and software architecture, there are some more concrete questions that have guided our research:

- How should a proper process be designed, both at a high level and in terms of concrete practices?
- How can the existing systems be analyzed and a future system outlined, rapidly and early enough while being at a sufficient level of detail to enable a well-founded decision?
- To what extent are the suggested practices unique to the context of in-house integration?
- To what extent are these practices today employed successfully, and to what extent are they overlooked?

The more specific questions have in each research phase been further guided by the following three types of (sub-)questions, which at the same time describe micro-steps of the research method:

1. **Survey Existing Practice.** What ways of working are used in existing organizations?
2. **Evaluate Existing Practice.** What are the experiences of these organizations? In their own opinion, what mistakes did they make, and what were they successful with?
3. **Generalize.** To what extent can these experiences be generalized into suggestions for other organizations?

1.3 Research Phases and Methods

In general, the type of study method to be used depends on the research problem and the maturity of the research field [313,347]. Exploratory studies are needed for new problems where there are no developed theories and where not even the concepts to study are very well known. As knowledge about the problem is gathered and theories are developed, the research would turn towards theory validation in the form of e.g. replicated experiments and statistical methods. In the early stages, studies are more of a qualitative nature, while later studies aim at quantifying the subject studied.

These general observations describe the research of this thesis well. As described in the section on research questions (Section 1.2), the research began with a survey of the current state of organizations, and their own evaluation of how successful they have been. These experiences have then been generalized to give guidelines. According to the series of study questions, the research has progressed through five clearly distinguishable research phases. The three types of questions described above – survey, evaluate, and generalize – are also clearly identifiable within each phase. Through participation in an industrial case (phase one), followed by a thorough search for related existing publications (phase two), I realized that in-house integration is a new and relevant topic to be studied on its own. Experience from more organizations was collected (phase three), this leading to two follow-up studies: one studying *Merge* more closely (phase four), and one validating and quantifying the previous findings (phase five). This sequence of research phases is depicted in Figure 1.

The rest of this section introduces the phases briefly, each in its own paragraph. The research method of each is described in depth in section 3.3 and the complete published results of each are given in the appended papers and appendices.

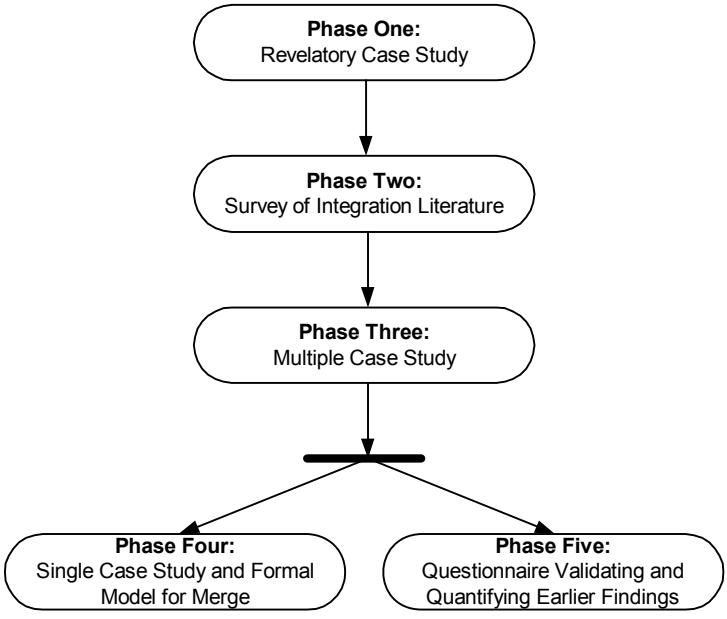


Figure 1. Research phases.

Phase One: Exploratory Case Study. I had the opportunity to participate in an industrial project, in which three systems within a newly merged company were found to have a similar purpose. Users and architects met to evaluate the existing systems and outline possible alternatives for an integrated system, including the possibility of discontinuing some of the existing system(s). Management was then to agree upon the implementation of one of these solutions. I obtained the data used as a participant in the project. The questionnaire was used to obtain the experiences and opinions of some of the other participants (the questionnaire form and collected data are reprinted in Appendix A). The findings should be considered as lessons learned from a single case, illustrating a topic not previously researched as such. The three publications that resulted are to be seen as experience reports [207,211,217]. Two of these publications are included in this thesis as Paper I and Paper III. The events of this case were also further discussed in my licentiate thesis[3] [206].

[3] The Licentiate degree is a Swedish degree somewhere between a M.Sc. and a Ph.D. degree.

Phase Two: Survey of Integration Literature. In phase one, it was difficult to position the case study in relation to existing literature. A major survey of the relevant literature was performed to investigate to what extent the case experiences appeared in existing research publications. The relevant literature had been searched for and consulted both before and after, but for this phase, a systematic search scheme was designed. Publications containing certain keywords were searched for in publication databases, book lists, etc. Many publications were discarded on the basis of title and abstract, but many were screened, and many publications studied more thoroughly. An exhaustive search for new information was made in the literature studied more thoroughly. This literature survey resulted in one publication [212], which has been re-worked and extended into Section 4.1. This phase enabled the formulation of the in-house integration of software systems as a largely unexplored research challenge.

Phase Three: Multiple Case Study. Based on the first two phases, a set of open-ended interview questions were formulated (reprinted in Appendix B) and an active search was made for more cases with experience from in-house integration projects. No theory had been developed at this stage but various questions concerning the integration process, with a particular focus on technical characteristics of the systems, were asked. I studied nine such cases, mainly by performing interviews. Several data points enabled some general conclusions to be drawn concerning important issues to evaluate early in the integration process and the effects of not doing so, as well as some concrete practices and risk mitigation tactics. This phase resulted in five conference publications [209,214-216,220], both process related [209,214,220] and architecture related [209,215,216]. These were later combined and extended into one journal paper [213], which is included as Paper II in the thesis. This phase led to two separate research directions, as phases four and five.

Phase Four: Single Case Study and Formal Model for Merge. One observation made during phase three was that a very tight *Merge*[4] seemed to be the strategy with the most variants and being the most difficult to implement successfully. I therefore decided to study this particular strategy in more depth and returned to one of the cases in phase three, where I conducted follow-up interviews (the interview questions are reprinted in Appendix C). A method for rapidly exploring *Merge* alternatives has been

[4] Details about how we use this term can be found in section 2.1.

devised on the basis of this data. A prototype software tool to support the method has also been developed with the help of students. This phase resulted in one conference publication [210] describing the method itself and one workshop publication [218] describing the tool, which are included as Papers V and VI.

Phase Five: Questionnaire Validating and Quantifying Earlier Findings. As the multiple case study of phase three had led to a number of qualitative observations, a natural continuation of the work was to design a study aimed at validating these. In addition, there were many observations on the same level – such as an unordered list of suggested practices – which it would be useful to rank in importance. A questionnaire consisting of a number of questions with five-grade scales was therefore designed. The questionnaire was distributed to six of the previous cases and two others. (The questionnaire form is reprinted in Appendix D and the collected data in Appendix E.) The responses were analyzed and published as a conference publication [222] which is included as Paper IV.

1.4 Thesis Overview

Figure 2 describes the conceptual architecture of this research. There are *research questions*, which are studied in *research phases* – each using some *research method* – which result in *research results* as reported in *research papers*. *Related work* is important both when defining the questions and when reporting the results in papers.

The thesis is organized in the following way: a chapter or section is dedicated to each of these concepts, with extensive references to the others. Section 1.2 describes the research questions of the work. Section 1.3 presents an overview of the goals, research methods, and resulting papers of the five research phases. Chapter 2 describes the research results, by recapitulating the research questions, and shows how the research papers answer these questions. Chapter 3 discusses the validity of the results, and Chapter 4 surveys related work. Chapter 5 summarizes and concludes the thesis, followed by a list of references on page 71. This is followed by the research papers, reprinted with only layout changes; this means that each appended paper contains its own sections on related work, research questions, results, and references, all of which to some extent overlap earlier sections of the thesis.

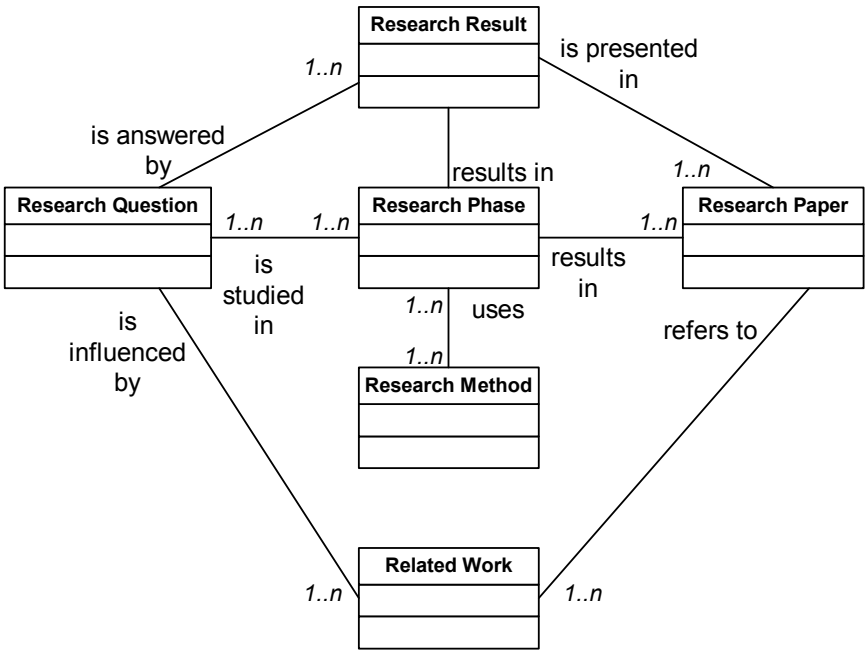


Figure 2. The concepts of the thesis and their relationships.

Chapter 2. Research Results

This chapter provides a brief overview of the research results, the details being presented in the appended papers. Figure 3 is a high-level overview of the results showing the different elements of a proposed integration process. There are two phases or sub-processes: a *vision process* (which results in a *decision*) and an *implementation process*. Of these two, the thesis focuses on the vision process, which involves the consideration of various *strategies* for the final system and their associated *project plans*. To be able to decide which strategy to implement, we describe the important elements of an *architectural analysis* as well as some *considerations concerning the retirement* of the existing systems. We have also observed a number of *practices* that should be employed in the integration process, i.e. some characteristics of the process at a fairly detailed level.

We have here aimed at outlining the main lines of thought and relating to each other the results in the different papers. We therefore provide extensive references to details in the included papers. We use italics for terms and concepts that are used and explained further in the appended papers. Section 2.1 describes most of these concepts at a fairly high level, section 2.2 presents the suggested practices, and section 2.3 describes the architectural analysis to be performed. This chapter concludes with section 2.4, in which the papers included in the thesis and the contributions of each paper (in particular mine) are listed.

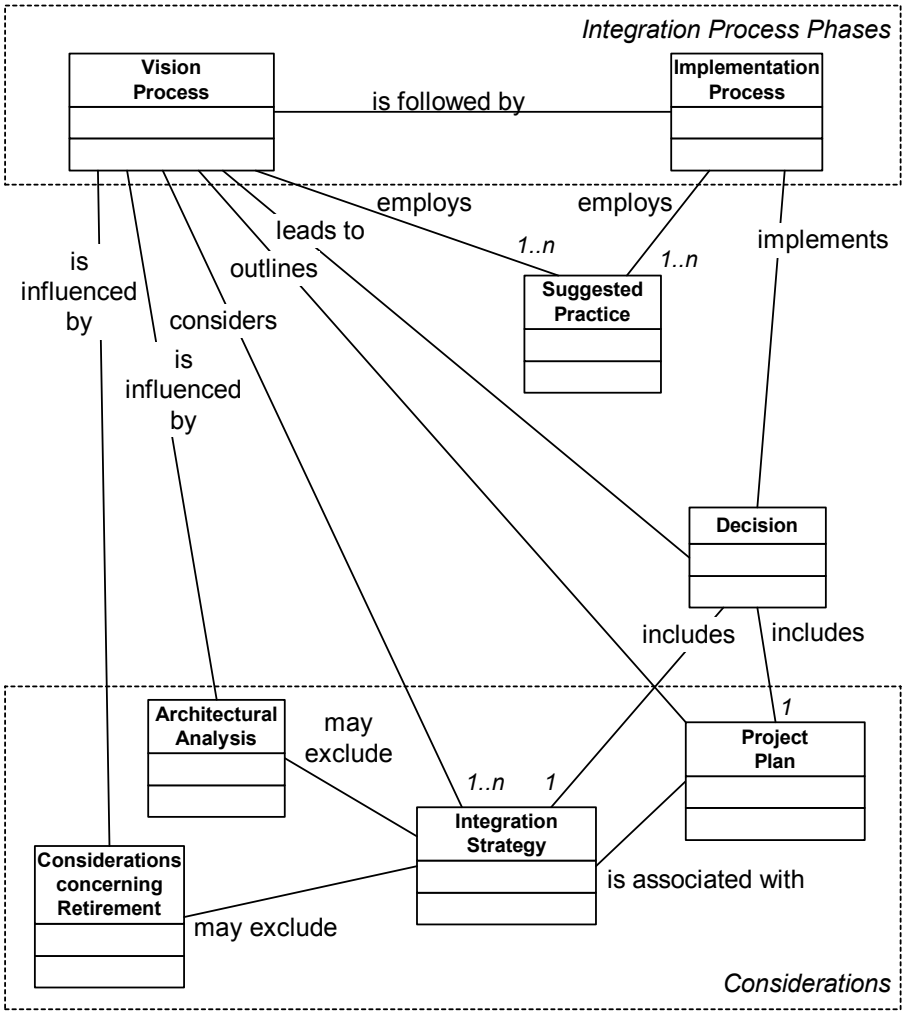


Figure 3. The important elements of the proposed integration process.

2.1 Process Model for In-House Integration

In-house integration is typically initiated by the senior management, as a result of an intention to rationalize (Paper II, section 3). In the integration process, it is possible to distinguish between a *vision process* and an *implementation process*. Even if this division is not always explicit, there is a clear difference between the purpose of each sub-process, the participants

in each, and the activities included in each (Paper II, section 1.2; Paper III, section 2). The vision process leads to a decision to a plan that includes a high-level description of the future system both in terms of features (requirements) and design (architectural description), as well as a project plan for the implementation process, including resources, schedule, deliverables, etc. (Paper II, section 1.2; Paper III, section 2). The target system could preferably be characterized in terms of the features of the existing systems, since these are well-known to the stakeholders (Paper II, section 3.2; Paper III, section 2; Paper IV, section 3.5; Paper V, section 2.3.1; Paper VI, section 2.1). The implementation process then consists of the execution of the plan.

At a high level, it is possible to distinguish between four strategies, characterized by the parts of the existing systems that are reused (Paper II, section 1.2; Paper IV, section 3.1): *Start from Scratch*, *Choose One*, *Merge*, and – to be comprehensive – *No Integration*. By introducing these idealized strategies, discussions can focus on two particular concerns that may effectively exclude one or several strategies: the architectural compatibility of the systems, and considerations concerning retirement (Paper II, Section 3.1; Paper IV, section 3.4). Of these two concerns, architectural compatibility is easier to describe objectively and correlate with the chosen solution; the existing systems being built the way they are, while the considerations concerning retirement involve business considerations and many stakeholders' opinions (Paper II, sections 4.3 and 5; Paper VI, section 3.4). Based on the findings, a simple checklist-based procedure has been developed, which ensures coverage of the main issues to be analyzed in order to understand the consequences of each potential strategy (Paper II, section 8.2) – even if, as is common, an outlined alternative lies somewhere between these idealized strategies (Paper I, section 4; Paper II, section 1.2, 2.2 and 8.1; Paper IV, section 3.1.1).

For *Choose One* and *Start from Scratch*, one must consider the impact of retirement (Paper II, section 5). Two influential factors when considering the feasibility of retirement are the *stakeholders' satisfaction with the existing systems* and the *life cycle phase of the existing systems* (Paper II, section 5.1). For *Choose One*, one must also estimate the degree to which each of the existing systems would replace the others, by considering different stakeholders' points of view (Paper I, section 4; Paper III, sections 2 and 3). Typically, if a system is replaced by another, there is a need to ensure backward compatibility and provide migration solutions (Paper II, sections 5.2 and 7).

The *Merge* strategy means reassembling parts from several systems into a new system, and the most important issue to analyze is the compatibility of the systems (see section 2.3 below). When considering the *Merge* strategy, the procedure becomes recursive, so that for each component in the systems it is possible to discuss whether to *Choose One*, or *Start from Scratch* and create a new component, or *Merge* the components by decomposing the components; the same types of analyses (i.e. impact of retirement, compatibility, etc.) must be performed for these alternatives (Paper II, sections 4.1 and 4.3).

An implementation plan must be outlined for the selected strategy, considering resources available and what costs and risks would be acceptable (Paper I, sections 4.2 and 4.3; Paper II, sections 6 and 7). The characteristics of the plan will depend on the strategy selected. For *Start from Scratch*, the plan must take into account the development and deployment of the new system, and for *Choose One*, the evolution and deployment of the chosen system (Paper II, section 6). For both of these strategies, the challenges of the required parallel maintenance and eventual retirement of (some of) the existing systems must also be addressed (Paper II, section 6) as well as the additional costs of migration solutions (Paper II, sections 5.2 and 7). For the *Merge* strategy, stepwise deliveries of the existing systems should be planned, thus enabling an *Evolutionary Merge*, and the complexity of the parallel maintenance and evolution of the existing systems must be taken into account (Paper II, section 6). For the *Merge* strategy, there is often a difference between the time scale and complexity envisioned by the senior management, which could be labeled *Rapid Merge*, and an *Evolutionary Merge* (Paper I, section 4.2; Paper II, sections 1.2, 2.2, and 8.1). The *Merge* strategy requires a longer period of distributed development and a need for synchronization, and results in potential conflicts between local and global goals and prioritizations at different sites (Paper II, section 6.3).

2.2 Practices

A number of beneficial practices have been identified. Some were encountered in the single case study of the first phase of the research (Paper III, sections 2 and 3), but only identified as such and further described in the multiple case study in phase three (Paper II, sections 3.2 and 6). Their relative importance was indicated by means of a questionnaire in research phase five (Paper IV, section 3.5).

During the vision process, two closely related practices were identified: to assemble a *small evaluation group* and *collect experience from existing systems* (Paper II, section 3.2; Paper III, section 2). Although these are good practices in many software activities, they seem to be particularly important during in-house integration projects; this is because a collective overview of the systems must be obtained, and the previously separate groups of people now need to cooperate (Paper II, section 3.2; Paper III, section 2). Various stakeholders should evaluate the existing systems from their respective points of view, and the requirements on the future system should preferably be stated in terms of the existing systems, in order to reuse the results of the requirements elicitation already performed for the existing systems, as well as to evaluate the existing implementations of these requirements (Paper II, sections 3.2 and 4.1; Paper III, sections 2 and 3; Paper V, section 2.3.1). In the study, these two practices have been considered among the most important of all practices, but have usually not been implemented to the extent they should (Paper IV, section 3.5). Mechanisms and roles must be defined in a way that ensures that a *timely decision* can be made in spite of stakeholders not agreeing completely (Paper II, section 5.2). Stakeholders will probably not be satisfied with a costly and time-consuming systems integration that in the end will only present them with the same features presented by the existing systems; it is therefore necessary to *improve the current state* so that the future system is an improvement of the existing systems (e.g., has richer functionality or higher quality) (Paper II, section 3.2). Another practice considered important – somewhat contradicting the need for *timely decisions* – is to perform a *sufficient analysis* (Paper II, section 3.2). Based on the current data it is not possible to determine which of *timely decision* or *sufficient analysis* is in general more important for in-house integration (Paper IV, section 3.5).

During the implementation process, *commitment* is very important (Paper II, section 6.1; Paper IV, section 3.5). In particular, a *strong project management* is needed, but success also depends on *cooperative grassroots* (i.e. the people who will actually do the hard and basic work) (Paper II, section 6.1; Paper IV, section 3.5). These aspects are frequently overlooked (Paper IV, section 3.5). The most important aspect, and the most often overlooked, is that management needs to show its commitment by allocating sufficient and adequate resources (Paper II, section 6.1; Paper IV, section 3.5). Another practice very often overlooked is to *make agreements and keep them*, this in a more formalized manner than the (previous) organizations are accustomed to (Paper II, section 6.1; Paper IV, section 3.5). This may be because the challenges of distributed activities have not been encountered

before in the organization(s) and are not well known, and/or because of a strong reaction from staff as soon as the retirement of "their" system is even remotely considered (Paper II, section 6.1; Paper IV, section 3.5). A *common development environment* is needed, i.e. infrastructure support for e.g. dividing work and sharing development artifacts, a common set of development tools etc. (Paper II, section 6.1; Paper IV, section 3.5).

Due to the long time scale of especially the *Merge* strategy (since the *Rapid Merge* seems not to be a realistic alternative), a *stepwise delivery* approach should be employed, so that the existing systems can still be delivered several times in the short term, while the long-term goal is a merged system (Paper II, section 6.2). In order to succeed with this, one must find ways of *achieving momentum* in the integration process, by implementing changes that will achieve the long-term integration goal and which are also useful in the short term; making such changes to the systems will, to some extent, contribute to their more rapid convergence (Paper II, section 6.2).

2.3 Architectural Analysis

The findings and understanding concerning architectural analysis have evolved and been refined through all the research phases, from initial *observations* and *lessons learned* [300,348] in phase one (Papers I and III) to include a broader, generalizable source of experiences in phase three (Paper II), some reasoning about how to perform an analysis in order to explore various *Merge* alternatives in phase four (Papers V and VI), and validation of these findings in phase five (Paper IV).

As described above, there is typically no single individual having technical knowledge of all existing systems (Paper II, section 3.2; Paper III, section 2). To enable rapid analyses, the technical features of the systems need to be discussed at a high, i.e., architectural level. The first step is, therefore, to prepare a common ground for discussion, which for architectural analysis means that similar architectural descriptions need to be created (Paper I, sections 4 and 6; Paper III, section 2; Paper V, section 2.3.1; Paper VI, section 2.1). This makes it possible to discuss known strengths and weaknesses of the existing architectural design solutions, and the possibilities of reusing individual components (Paper I, section 4; Paper II, sections 4.1 and 4.3). From these architectural descriptions, it is possible to design alternatives of a future system, which can be evaluated from different points of view given that relevant properties of the components are annotated (Paper I, section 4; Paper III, section 2; Paper V, sections 2.3.2 and 3.2;

Paper VI, section 2.2). For example, if each component is annotated with the estimated effort required for its modification, it is possible to calculate an approximation of the (minimum) total implementation effort (Paper I, section 4.2; Paper V, sections 2.3 and 2.3.2; Paper VI, section 2.2). It is also possible to evaluate future maintenance efforts, measured by the number of technologies used, program size (LOC), and conceptual integrity (Paper I, section 4.1). Quality and features can be discussed both component by component (i.e., considering which of two alternative components is the more desirable) and at system level (i.e., considering the system level qualities) (Paper II, section 2.2; Paper V, sections 2.3.2 and 3.2).

The more incompatibilities between the existing systems are found, the less feasible it becomes to consider reassembling components and make them work together (Paper I, sections 4 and 4.1; Paper II, section 4.3; Paper IV, section 3.4). The studies have enabled the identification of three high-level aspects of architectural incompatibilities, which are likely to cause problems if the differences are too large: structures, frameworks, and data models (Paper II, section 4.4). Based on the studied cases, there is convincing evidence that the structures of the systems must be very similar for it to be feasible, in practice, to *Merge* them (Paper II, section 4.3). In this context, "framework" should be understood broadly, as "an environment that defines components", i.e. an environment specifying certain rules concerning how components are defined and how they interact (Paper II, section 4.1); the observation here is that interfaces (in a broad sense, including, for example, file formats, API signatures, and call protocols) must be similar in format and semantics for *Merge* to be feasible. An exact match is not however necessary since it is always technically possible to modify the systems (Paper II, section 4.1; Paper IV, section 3.3). Since data is processed and interpreted in many parts of the system, too large differences between the data models of the systems means that the *Merge* strategy is practically infeasible (Paper II, section 4.4).

In the cases studied, at least, the systems to be integrated often exhibited certain types of similarities and are thus not as incompatible as one would perhaps expect: technologies and programming languages are often similar or the same, and it is not uncommon that a particular technology is used to support a componentized architecture (Paper II, sections 2.2 and 4.3; Paper IV, section 3.3). The systems very often have components with similar roles but these components may be structured in different ways; the most similarities can be expected between hardware topologies (Paper II, section 4.4; Paper IV, section 3.3). Existing user interfaces also show some amount of similarities (Paper IV, section 3.3). Similarities can often be traced to the

time when the first systems of a certain type were created, which means that certain ways of solving certain problems have become cemented in a number of systems which are still in use (Paper II, section 4.3; Paper IV, section 3.3). There are often, also, some domain standards applicable to the systems, which make them similar in at least some respects (Paper II, section 4.3; Paper IV, section 3.3). We also found an additional, rather unexpected source of similarities: the systems may have been evolved independently (i.e. branched) from a common ancestor (Paper II, section 4.3; Paper IV, section 3.3). To formulate these observations as a guideline: if the systems address essentially the same problem, and/or if they are contemporaneous, and/or if there are standards within that particular domain, and/or if the existing systems have some common ancestry due to previous collaborations, the systems are possibly similar enough for the *Merge* strategy to be seriously considered.

2.4 Summary of Included Papers

This section describes the results of each appended paper in terms of the results described above, and indicates my personal contribution of each paper.

Paper I: "Software Systems Integration and Architectural Analysis – A Case Study", Rikard Land, Ivica Crnkovic, *Proceedings of International Conference on Software Maintenance (ICSM)*, Amsterdam, Netherlands, September 2003

This paper describes *observations* and *lessons learned* [300,348] from the single case study of phase one. Here we can find some fundaments of the integration process, architectural reasoning (section 4), and an early characterization of integration strategies (section 3).

I was the main author; I participated in the case study as an active project member, making observations and submitting reflections. My supervisor and coauthor was a valuable mentor, and both authors related the case study to existing research literature, and formulated general conclusions.

Paper II. "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", Rikard Land, Ivica Crnkovic, accepted for publication in *Journal of Information and Software Technology*, Elsevier, 2006

This journal paper describes the multiple case study of phase three and provides an extensive analysis and synthesis of observations from nine cases of in-house integration. The paper describes the overall process, integration strategies, architectural analysis and the role and sources of architectural incompatibility, important considerations regarding the retirement of existing systems, other issues to evaluate, and observed practices. This paper builds on several earlier conference publications [209,214-216,220].

I was the main author leading all phases of the study. Early design and analysis was performed with the help of my supervisor and coauthor (as well as other colleagues, co-authors of the earlier conference papers). During the writing process, my supervisor and coauthor have made many suggestions and given much advice, and we have had many constructive discussions.

Paper III: "Integration of Software Systems – Process Challenges", Rikard Land, Ivica Crnkovic, Christina Wallin, *Proceedings of Euromicro Conference, Track on Software Process and Product Improvement (SPPI)*, Antalya, Turkey, September 2003

This paper describes the case study of phase one, focusing on overall process characteristics and certain practices. It can be read as an in-depth example of the *small evaluation group* practice.

I was the main author; I participated in the case study as an active project member, making observations and submitting reflections. The coauthors aided in relating the case study to existing research literature and formulating general conclusions.

Paper IV. "Software In-House Integration – Quantified Experiences from Industry", Rikard Land, Peter Thilenius, Stig Larsson, Ivica Crnkovic, *Proceedings of Euromicro Conference Software Engineering and Advanced Applications, Track on Software Process and Product Improvement (SPPI)*, Cavtat, Croatia, August-September 2006

This paper reports the results of phase five. Based on a questionnaire survey, the paper quantifies and validates some of the earlier qualitative findings: various aspects of architectural compatibility, decision making considerations, integration strategies, and practices.

I was the main author; my contribution being to lead all phases of the study. The coauthors were involved in the outlining of the study,

discussions during its execution, the designing and distribution of the questionnaire, the analysis of the results, and the writing of the paper. Peter Thilenius stood for the expertise concerning questionnaire design and statistical analysis.

Paper V. "Merging In-House Developed Software Systems – A Method for Exploring Alternatives", Rikard Land, Jan Carlson, Stig Larsson, Ivica Crnkovic, *Proceedings of the 2nd International Conference on the Quality of Software Architecture*, Västerås, Sweden, June 2006

This paper is based on a follow-up study of a case which implemented the *Merge* strategy. The paper suggests a method for exploring various *Merge* alternatives, by making incompatibilities explicit, recording decisions made, and guiding the exploration on the basis of information entered. The method is designed to be used by a *small evaluation group* of architects.

I was the main author; I led the study and conducted the case study interviews. Jan Carlson and I took the method from initial idea to a formalized method, where Jan stood for the expertise in formal modeling. The other coauthors were involved in outlining the study and discuss it throughout.

Paper VI. "A Tool for Exploring Software Systems Merge Alternatives", Rikard Land, Miroslav Lakotic, *International ERCIM Workshop on Software Evolution*, p 113-118, Lille, France, April, 2006

This paper describes a tool supporting the method described in Paper V.

I was the main author; my contribution being to act as customer and steering group for a student group in a university course project which implemented the tool. One of the students, as coauthor, assisted in the writing of the paper and further updated the tool after the course had ended.

Chapter 3. Validity of the Research

Why should the results of this thesis be accepted? And how general are they? These are important questions, and are not easily answered. The goal of this chapter is to show that the results have been achieved by systematic study and that an amount of *external validity* has been established for the results.

In the research field of Software Engineering, several research traditions and methods meet. Here we find mathematical reasoning alongside studies of human behavior, technology, business, society, and their interaction. Quantitative studies are performed in parallel with qualitative research, purely theoretical and analytical reasoning with highly pragmatic observational studies. There is no single articulated research tradition to adhere to, no commonly agreed upon guiding rules for conducting and evaluating research, no consensus on what makes a study "scientific" and "valid" [348]. This chapter therefore begins by briefly reviewing various research traditions and views of science (Section 3.1), and continues by describing the most relevant research methods (Section 3.2). Since external validity (the ultimate goal) requires that construct validity, internal validity and reliability are achieved, the larger part of the chapter describes in detail how the research has been carried out (section 3.3). Section 3.4, which concludes this chapter, is a synthesis of these accounts, and discusses to what extent the results are externally valid.

3.1 Research Traditions

There are a number of research traditions, of which those most influential in shaping the field of Software Engineering are briefly described here. We do this because the meaning of validity may be rather different in different traditions.

3.1.1 Characterizing Science

In *empirical science* essential elements are *theories*, which engender *predictions*, which can be correlated with *observations*. Traditional criteria for evaluating this type of research include issues such as the objectivity of the researcher[5], systematic and rigorous procedures, the validity of data, triangulation, and reliability [300]. However, even a high number of observations cannot "prove" a theory right, only "support" it; an essential element of a scientific theory is, therefore, that it must be *falsifiable* [63,308]. The commonsense *inductive* argument says that the more supporting data, the stronger supported the theory is. However, this standpoint is difficult to defend logically [63,308], and an alternative is the notion of corroboration [308], which means that a theory must have withstood a number of tests aimed at falsifying it, or comparing it with a competing theory. However, there are some limitations both in principle and practice. First, empirical science is most suitable when the subject of study lends itself to relatively simple, quantifiable models. Also, observations are subject to e.g. measurement errors, inappropriate use of measurement instruments – which may be inadequate in any case – and not least, predispositions of the observer making the observations [63,76]. When observations contradict the theory there is no way to deduce with logic alone where the error lies – in the theory, the observation, or in some additional assumption or theory [63]. Historically, this has caused numerous controversies between competing theories, in which the proponents of each side disqualify the other's observations and experimental settings [76]. For all these reasons, one must be careful to distinguish between observations and facts[6].

Naturalistic enquiry means to study the real world, where the researcher does not attempt to manipulate the phenomenon of interest – as opposed to an experimental setting [300]. This is typical for social sciences and is

[5] Total objectivity may be a too idealistic view; however, the researcher should strive to maintain some scientific integrity with respect to various interests that could bias the results, and define, follow, and document research procedures that could in principle have been used by someone else.

[6] All these arguments should make us careful in attempts to distinguish "science" from "non-science" [63]. Taken somewhat to the extreme, these arguments have led to *deconstructive* and *relativistic* standpoints, according to which science is mainly a social activity (i.e. scientists have achieved a certain status), and it is consequently meaningless to discuss such a thing as validity.

common in Software Engineering when it comes to studies of the social and psychological aspects of software, such as usability [280,374] or the introduction of a new process or method into a development team [192].

There is an element of *interpretation* involved in most or all research, including Software Engineering, and consequently also this thesis. The *hermeneutic* research tradition emphasizes the interpretative element, and is the prevalent tradition in studies of e.g. literature, law [300,392]. The notion of *text* can be extended beyond written texts to include speech, multimedia, or any occurrence. In the hermeneutic tradition, there is little sense in discussing external validity; validity here rather means a reasonable explanation which appeals to universal human experiences and provides an understanding of the artifact studied (see further discussion under 3.1.2 below).

Computer Science is largely founded on *logics and mathematics*, in which there are no observations of an external world [372]; validity here means formal correctness. Computer Science and formal models are an important part of Software Engineering, but here the focus shifts from correctness towards usefulness in an engineering context (i.e. closer to naturalistic enquiry) [192,332].

Ethnography takes a cultural perspective [300], and has found its way into Software Engineering [321]. The traditions of *phenomenology* and *social construction* (and *constructivism* in general) would also be interesting to apply in Software Engineering, as they focus on people's experiences and how they explain and "construct" the world they inhabit [300,392]. Other traditions include the *positivist* and *realist* traditions, but these seem less influential in Software Engineering as their primary focus is on the notions of *reality* and *truth* [392]; in Software Engineering we are more interested in usefulness (in this sense our research field belongs to the *pragmatic* tradition).

Historical explanations of how science progresses adds an interesting perspective to the discussion about validity (e.g. conformance to a *paradigm* in *normal science* [76,202]) but are of no help for individual researchers or individual studies [63], other than making us humble about the validity of our studies.

3.1.2 Quantitative and Qualitative Research

It is important to distinguish between *quantitative* and *qualitative* research. Which one to choose depends on the purpose of a particular study: quantitative studies can give a certain amount of precision in a mathematical sense, but require the question to be studied to be well-understood and appropriate measurement instruments to be available (cf. the discussion on empirical science in 3.1.1). A qualitative study should be chosen when the research question is more open, when the topic being studied has, as yet, no strong theory that guides the design of the study, when the context cannot be separated from the phenomenon being studied, and/or when individual personal experiences of the phenomenon are as important as the phenomenon itself [300]. Since in qualitative studies the researcher has less firm theory on which to base the study design, these kinds of studies are usually more flexible as the research unfolds naturally and new opportunities for observations appear. For this reason, the terms *flexible* and *fixed designs* are sometimes used instead of the quantitative-qualitative dichotomy [313]. Many study questions, not least in the field of Software Engineering, are multi-faceted and thus must include both quantitative measurements and qualitative data [300].

Four types of validity commonly referred to are: construct validity, internal validity, reliability (or conclusion validity), and external validity (or generalizability) [313,395,403]. (These are further discussed in section 3.3.) These types of validity are applicable to both quantitative and qualitative research, and the first three in particular are closely connected with the traditional evaluation criteria for research such as researcher objectivity, systematic and rigorous procedures, and triangulation [300]. When considering the final goal of a study, and its external validity, there are differences between quantitative and qualitative research. Quantitative research has a theoretical foundation in statistics, in which terms such as *probability* and *confidence* have a well-defined mathematical meaning [276]. External validity is achieved by showing that the prerequisites are fulfilled (i.e. the population is well defined, some appropriate sampling strategy has been chosen, etc.). Although this to some extent is also applicable to qualitative research, it has been argued that *understanding* of the phenomenon studied – as judged by others – is ultimately the only validation possible [254]. If people consider an explanation to make sense, i.e. if it actually explains something to them, it should be considered valid (cf. the discussion on interpretations and hermeneutics in 3.1.1). For complex occurrences considerably dependent on their social and economical contexts

(including places and points of time), there are more or less reasonable ways of explaining phenomena, but labels as "right" or "wrong" are not appropriate. Conclusions are made interesting for some group of people [29]. "Scientists socially construct their findings." [96] However, validation cannot be totally arbitrary; any claim needs to be strongly supported by data and the reasoning that led to a certain conclusion [254]. I agree that "insight, untested and unsupported, is an insufficient guarantee of truth." [320]

3.1.3 Positioning This Thesis in the Context of Research Traditions

The research presented in this thesis is mostly in the form of naturalistic, qualitative, flexible, observational studies (phases one, three, and four). It has also involved a formal model (phase four), the usefulness of which however remains to be validated. The fifth phase aims at quantifying earlier results to some extent. All phases contain an interpretative element, and there is an implicit inductive argument in that similar phenomena are observed in several cases, and also since some of these observations are similar to those of others. Concerning validation, the goal of the thesis is to provide a certain amount of insight and understanding of software in-house integration rather than to present quantitative results based on statistical analyses. The details concerning construct validity, internal validity, and reliability are presented in section 3.3 in order to show how the thesis fulfills the traditional criteria for quality research.

3.2 Relevant Research Methods

Let us now turn to a more concrete level and look at various research methods, in order to motivate the choice of method in each research phase.

The goal of a research study is often to establish a relation between certain variables; some are controlled as part of the study setup (called *independent variables*), and some output (*dependent variables*) are recorded. When a theory is to be tested, the outputs are correlated with predictions. Depending on the area of study, and the specific questions, it may be difficult to control (or even measure) the input variables, and different research methods are thus suitable in different situations. Also, depending on how mature a theory is, different kinds of tests are needed. Initially, some sense is required to be made out of seemingly chaotic data, after which a theory is formulated.

There is first a focus on gathering some support and only later on testing the theory through falsification attempts, or comparison with rival theories [63,348].

This section describes some common research methods and the circumstances under which they are suited, and then motivates the choices of research methods in the five phases.

3.2.1 The Case Study

For contemporary problems which cannot be properly studied outside their different complex contexts – and where the complete context may not even be known – the *case study* [403] is suggested as an appropriate research method. A *multiple case study*, i.e. a study of several cases with known similarities and differences, is considered to give a higher confidence in the external validity than a single case study [403]. A *single case study* is appropriate for example when a research question is new, when a case has such properties that it would put the theory to a severe test (a *critical case*) or when a certain case is thought to be extreme in some other way, such as a successful (or disastrous) project, which would be a good source from which to learn (an *extreme case* or *illuminative case*) [300,403]. Time and resource limitations might also prohibit more than one case to be studied. A *revelatory case* is one, the importance of which is only realized by the researcher during (or after) the study, for example in characterizing a new research problem [403]. Often the results of case studies are reported as *observations* or *lessons learned* [300,348]. If a case study is planned so that a contemporary event is studied when it occurs, it is possible to perform the same measurements before and after the event – which is an advantage from a scientific point of view. In some case studies, however, the chain of events being studied is partly historical, as for example when it is only realized after some initial events that it is worth being studied {Yin 2003 867 /id} – such as the topic of in-house integration.

The problem with case studies is that the complex context, in terms of many influential (partly unknown) factors, makes it difficult to generalize the results. This is of course not a problem if the purpose is indeed to evaluate something for use in a particular context (for example within a specific organization) [192], but to be able to claim any wider external validity, the best advice available is to propose and evaluate several *rival theories* as explanations of the results [300,403]. And as explained in the discussion

about qualitative studies (section 3.1.2), an important goal is to provide understanding, i.e. an explanation that others find reasonable [29,96,254].

3.2.2 Grounded Theory Research

According to the grounded theory research method [359], theory is constructed from data even if the researcher has only few and vague preconceptions of the problem under study. With this method, data is collected, leading to the proposal of some initial theory. Data collection continues, guided by the theory, and after each round of data collection the theory is adjusted to explain the data collected so far. This continues in an iterative manner until a satisfactory level of agreement between new data and the theory is attained. This method aims at developing a new theory (which can be contrasted with the positivist ideal of empirical science, in which data should be collected in order to test a particular proposition formulated in advance). The grounded theory method originates in social science, and tries to account for some of the characteristics of that field: the important parts of the expected results are qualitative, and the data may be expensive to collect. It is necessary to be practical and efficient for larger scale studies so that the data to be collected for each new study object can be more accurately defined – guided by an analysis of the previously collected data – and thus collected more rapidly. The method has also found its way into Software Engineering and Information Systems [274,277,363]. Also, grounded theory research is typical for fundamental or basic research, in order to provide some insight into a phenomenon, but is not necessarily followed by action [300].

Grounded theory should not be mistaken for free-range exploration with no predispositions at all; this is seldom the case for a researcher [313]. Even without an explicit initial theory or proposition, or even a well-articulated research question, there is no such thing as a *tabula rasa* (“unscribed tablet”); the researcher will always be guided by his or her previous knowledge and experience [313]. In my opinion, the strength of the grounded theory method is that it codifies the element of an early qualitative study (when, as yet, there is no theory to be tested) in that it emphasizes a constant interplay between data and theory [274,300].

In a grounded theory study, it is difficult to claim external validity – the theory was built from a certain set of data and has not been tested on other data. As it is a qualitative method, the sought-after type of validity is (as described in section 3.1.2) an understanding of the phenomenon being

studied, which is (partly) argued for by demonstrating a rigorous approach. For studies in social sciences, where the grounded theory method originated, external validity is not always the goal, but the theory being built is (or should at least be) falsifiable in order to be scientific. Typically, for a theory developed this way, further studies are needed – employing other methods – in order to claim external validity. Of all qualitative methods grounded theory research is among those most in accordance with the traditional research criteria (e.g. objectivity of researcher, systematic and rigorous procedures, validity of data, triangulation, reliability, external validity) [300].

3.2.3 The Experiment

The classical method of empirical science is the *experiment*. The researcher typically makes several measurements while adjusting the independent variables, and records the output (dependent variables). This makes it possible to test theories rigorously in order to refute or support them (by comparing the values of the output variables with those predicted by the theory), or to determine the numerical value of a constant in a theory. The experiment has been a successful method in natural sciences and medical studies, and has found its way into Software Engineering [23,367,395,406].

For example, if one wants to determine whether the use of a certain process is better (in some sense) than the use of another, one could study a project group following the process and an equivalent project group following the other, and measure which was most more successful. Large-scale complex phenomena, which cannot be controlled by the researcher, can be studied in a *natural experiment* [300]. This means that the phenomenon is studied before and after a known, naturally occurring change in input parameters.

3.2.4 Formal Proofs

Mathematics and formal reasoning are essential tools for precisely formulating and analyzing concepts and ideas (see e.g. [1,6,78]). However, in Software Engineering the usefulness or feasibility of a concept (which must be studied using some other method) is equally important as its formal correctness.

3.2.5 Construction

The construction of software as a proof of some concept is common in Software Engineering research, often in the form of a tool supporting a process [90,91,102,177,179,248,325,326]. Seen in isolation, the scientific value of such construction can only be to prove that building this kind of software is possible – which may indeed sometimes be an achievement [348]. More interesting as a Software Engineering result is the evaluation of the tool in terms of feasibility, usefulness, efficiency, or performance.

3.2.6 Positioning This Thesis in the Context of Research Methods

As the topic of this thesis is a contemporary, complex phenomenon, research is largely based on case studies. In the first phase of my research, I took the opportunity to participate in a potentially interesting project, but was at that time not aware of the topic (in-house integration) for which I would later use the case study as an illustration (it is thus a *revelatory case*). There was no relevant theory or proposition, and the way forward chosen, the best available, was to collect further experiences (with a focus on architecture and processes) from organizations in a *multiple case study* in phase three. In phase four, one of the previous cases was selected for a new case study (concerning the *Merge* strategy) with a new set of questions. The case was chosen as an *extreme case*, the only one for which the *Merge* strategy had been clearly chosen and successfully implemented (although implementation is not completed yet).

As the research has progressed from a state of no proposition at all, data has been collected in order to build theory in a series of studies according to the grounded theory scheme. After the exploratory/revelatory case study of phase one, I performed a literature survey in phase two to formulate more precise questions for further data collection in phase three. This enabled the formulation of more specific questions, studied in phases four and five. Particularly within phases three and four, the data collection has been more directed as more data is collected (i.e. preliminary observations after a few interviews has led to more specific questions in the later interviews). So far, we have not developed a theory sufficiently to carry out an experiment, nor has there been instruments fine enough for measuring the outcome.

Data has been collected through project participation, direct observations, interviews, and questionnaires. In my studies of the literature, I have aimed

at being as rigorous and systematic as possible in defining, documenting, and following a protocol. A formal model has also been constructed, which has been implemented in a software tool; the usefulness and feasibility of these will be further validated in a real-life context, e.g. in the form of a case study or natural experiment.

3.3 Rigor and Validity in Each Research Phase

The rest of this section describes the research methods of each phase in detail. The motivation for this section is that to claim external validity, one must have achieved three other types of validity:

- **Construct validity** means ensuring that the data measured and used actually reflects the phenomenon under study. The general advice to achieve this is to triangulate data [96,300,313,403], i.e. to collect different types of data (e.g. both interviews and measurements) from several independent sources (e.g. interviewing more than one person). Yin also gives the advice of establishing a chain of evidence (i.e. documenting how conclusions made are traceable to data) and letting key informants review the draft case study report [403]. For interviews and questionnaires, construct validity also means that the researcher must also avoid leading or ambiguous questions [313].
- **Reliability** concerns the repeatability of the study. Ideally, any one studying the exact same case (not only the same topic) should be able to repeat the data collection procedure and arrive at the same results (although this is difficult in practice for phenomena that change over time). This is ensured by establishing and documenting how data is collected; Yin's two pieces of advice are to document and use a case study protocol and develop case study database where all data and metadata is collected [403].
- **Internal validity** means ensuring that the conclusions of the study are indeed true for the objects that have been studied, so that e.g. spurious relationships are not mistaken for true causes and effects [254,313]. Descriptions of data must be accurate, which can be ensured by introducing a review step where informants review e.g. copied out interview notes [313]. The researcher must also be open to different interpretations and theories, and avoid being predisposed to specific interpretations [254,313]. To increase the internal validity, there are several types of triangulation that should be employed [96,403]: data

triangulation (using more than one data point for the same observed data, e.g. using different people's opinions, studying the same object at different times), observer triangulation (using more than one observer to avoid subjectivism), methodological triangulation (using more than one method to analyze data), and theory triangulation (applying more than one explanation to the observations and compare how well each can explain the results).

That is, if a study uses the wrong indicators for the objects being studied (i.e. construct validity is not achieved), and/or is not internally valid, and/or is not replicable, it is not possible to claim external validity. Although a bit lengthy, this section is essential to motivate that I have been rigorous in following the available good practices in order to achieve these three types of validity. In addition, the characteristics of different methods have some direct implications on external validity as well, which is also described.

One difficulty, as pointed out in the introduction, is to judge whether a certain organization made the "right" or "wrong" decisions (if such things exist), whether they worked inefficiently or not, etc. Instead, the interviewees themselves have been asked to describe what they think should have done differently, what the most beneficial elements of their projects were, etc. My impression is that the respondents are well aware of whether they wasted time and money on activities that led nowhere, whether they were inefficient etc., based on their previous experiences from other projects and some general knowledge of good practices.

3.3.1 Phase One: Exploratory Case Study

I had the opportunity to be part of a project where a newly merged company had identified three overlapping software systems that addressed similar problems. The project would evaluate the existing systems from several points of view, identify some opportunities for creating an integrated system, and management would select one of the alternatives. My role was to aid the project leader in planning and documenting the project, and participating in discussions with the architects and developers of the systems. These discussions concerned both high-level decisions made in the systems, and two main alternatives for integration were outlined (plus the option of not integrating). In the end a decision was made for a loose integration. After the project finished, a questionnaire was distributed to the participants with some qualitative questions, which were then summarized in order to draw some conclusions in the form of lessons learned.

I had difficulties relating the case to existing literature on software integration, so its merit from a scientific point of view was that it illustrated a somewhat new relevant research topic. This is supported by the fact that the three papers reporting this case study were accepted for publication at three conferences, each paper describing the case from a different point of view: included as Paper I in the thesis is a description of the architectural analysis made [211] and as Paper III is a description of the process used [217]. In addition, we reported how the IEEE standard 1471-2000 [159] was used in the discussions on architecture [207].

Construct Validity

To ensure construct validity, data triangulation was achieved by using two different sources of evidence: personal participation in the project, and a questionnaire. The collection of six other people's points of view provided several data points. The reader may judge the quality of the questionnaire form itself, as it is reprinted in Appendix A together with the responses. I was careful to not make any speculative claims that are not founded in data, although the "chain of evidence" requested by Yin [403] was not explicitly constructed and managed, mainly due to my inexperience. One of the project participants (who participated in both the user evaluation and management's decision) read and commented the three papers describing the study, which is in line with Yin's advice of having key informants review the draft case study report [403].

The perhaps strongest criticism of the construct validity is that the decision was never implemented; this would mean that the case is not qualified as a good example. Part of the response is that the decision process itself (which is really the scope of the case study) should indeed qualify as a good example of a systematic process with certain analyses being made. (In retrospect, it seems clear that it was not possible to make a consensus decision that would effectively kill one or two of the three systems. And the organization never committed itself to implementing the decision, as it was considered inferior by the technicians.)

Reliability

The case study protocol was as simple as: participation in the project, followed by the distribution of a questionnaire. The participation experiences have not been documented as a data source, but questionnaire form and data, as well as the project documentation have been stored for future reference; the questionnaire form and data are reprinted in Appendix A but the project documentation is confidential.

Internal Validity

I participated in the project, and the other project members filled a questionnaire, so the data descriptions are accurate. As this case is used to illustrate a new research topic, it is difficult to argue that theory triangulation is applicable. How data triangulation was achieved (using multiple data sources) is discussed under "construct validity" above, but other types of triangulation (observer triangulation, methodological triangulation, and theory triangulation) was not implemented.

External Validity

As this was a single case study, the findings reported can be characterized as lessons learned from an interesting case. There is no formal foundation for claiming general applicability – the case may have been extreme and unique in some sense – but with some argumentation the experiences should be useful for other organizations as well.

3.3.2 Phase Two: Survey of Integration Literature

To really investigate whether my experiences were unique in published research, I made a thorough literature survey. Systematic literature review methods have been proposed for the purpose of finding evidence for a specific research question [190,356]. This involves creating a systematic protocol and documenting the search. In an exploratory search done in order to identify and profile a research field, there are some limitations with this type of reviews. A practical limitation is that the search terms cannot be very specific, and the very large number of hits must be filtered very rapidly. Another difference is that an exploratory literature review is qualitative rather than quantitative, which calls for interpretative and more creative analysis and argumentation. Nevertheless, being systematic and documenting the process, and implementing reviews by an additional researcher should increase the construct and internal validity as well as reliability of the literature review.

The conclusion was that no literature is to be found directly addressing the topic of in-house integration.

Construct Validity, Internal and External Validity

In this type of literature study, the topic being studied is the occurrence of publications on a topic, so construct validity seems not to be an issue[7]. Internal validity is satisfactory, as the distance is very small between the actual data (all referred literature) and the claim that the topic of in-house integration is little researched. General validity also seems not applicable to this type of study, as no general claims for certain types of objects are made.

Reliability

The study was to be systematic, so I prepared a reading list, empty at first. I planned the sites and databases where the search would start, and noted down the keywords I considered should lead to the relevant literature. For each search (one keyword in one database), the titles of all the hits were scanned, and most abstracts read. All hits that seemed interesting were added to the reading list. When actual papers were studied, all interesting references were added to the reading list. When the list was empty, this reading algorithm made me confident I had made an exhaustive search and found anything of interest.

The databases and sites searched were (in alphabetical order): ACM Digital Library [16], Amazon [8,9], CiteSeer [67], ELIN@Mälardalen (a search engine at my university, which searches several databases simultaneously), Google [121], IEEE Xplore [158], Kluwer and Springer journals [353]. The keywords used in the search were: “integration”, “interoperability”, “reuse”, and “merge”. The concrete result of this phase was a conference paper [212] with some one hundred references; this has been reworked into section 4.1 of the thesis. (I have no recorded figure of how many hits was actually scanned, neither of the total items in the reading list as it was continuously changing as items were added and removed; but there were hundreds of interesting hits, and due to the page limit for the conference paper not all of them were eventually used).

There is of course the possibility there is an important database with literature I have not been aware of, and which none of the papers found referred to. There is also a possibility that the wrong keywords were used,

---

[7] In other studies where databases are searched for information, ensuring construct validity may be more difficult, since is it is quite possible that the database does not accurately reflect the construct being studied (e.g., in a crime database one would not find all crimes in a given area and time interval, only the *reported* crimes).

that there is a body of knowledge with another terminology than what I was looking for. It is also possible that some important references were overlooked because of the human factor – I might have been very tired after scanning 499 titles so that I missed how promising the five hundredth would seem. I studied newer publications more carefully than older, with the motivation that I wanted to mirror the newest research. By searching mainly in article databases, textbooks are found only indirectly (via references in the papers found). It is therefore possible that older, seminal references, especially in the form of textbooks no longer in print, are missing.

3.3.3 Phase Three: Multiple Case Study

After the exploratory case study of phase one and the literature study of phase two, which together hinted at this being a new, relevant research topic, the most natural step was to continue collecting experiences from industry, and perform a broader study including several cases. As phase three, a multiple case study [403] was designed, where people in industry were interviewed (the questions are reprinted in Appendix B, and all copied out interview notes are found in a technical report [219]). All in all, 18 interviews in 9 different cases in 6 organizations were conducted. For each case, one to six open-ended interviews have been carried out with people deeply involved in the merge process and the systems, such as project managers, architects, and developers. In addition, some documentation has been available for some cases, and in one case (the same as in phase one) the author has been participating during two different periods. This study is the basis for Paper II. The cases are presented in Paper II, section 2.2 (and in more detail in the technical report [219]), and the rest of this section presents the research method, in particular how threats to the different types of validity were addresses. The text in this section is heavily based on the technical report [219].

Data collection was prepared by writing a set of interview questions, including a description of the purpose of the research (included as Appendix B). This was in most cases distributed to the interviewees in advance of the interviews, although in most cases the interviewee had not studied it in advance. The author prepared the questions, and two senior researchers reviewed these questions before the interviews started. Interviews has been considered the main data collection method, common to all cases, but as described earlier, other sources of data has been used as well when offered.

To collect the data, people willing to participate in the interviews were found through personal contacts. The interviews were to be held with a person in the organization who:

1. Had been in the organization and participated in the integration project long enough to know the history first-hand.
2. Had some sort of leading position, with first-hand insight into on what grounds decisions were made.
3. Is a technician, and had knowledge about the technical solutions considered and chosen.

All interviewees fulfilled either criteria 1 and 2 (project leaders with less insight into technology), or 1 and 3 (technical experts with less insight into the decisions made). In all cases, people and documentation complemented each other so that all three criteria are satisfactory fulfilled. Interviews were booked and carried out. Robson gives some useful pieces of advice concerning how to conduct interviews in order to e.g. not asking leading questions [313], which I have tried to follow. In some cases, the interviewees offered documents of different kinds (refer to the technical report [219] for more details).

Construct Validity

Multiple sources of evidence have been used as follows: for some of the cases, there are two or more interviews, and in some cases there are additional information as well (documentation, and/or personal experience with the systems and/or the organization). For others, one interview is the only source of information (which is clearly a deficiency). To some extent, this can be explained by the exploratory nature of the research, and also that the desire was to find a proper balance between the number of cases and the depth of each. The interviewees have also been invited to a workshop where pending results were discussed.

Reliability

Yin's [403] two pieces of advice have been followed carefully to ensure that someone could repeat the study:

- **Use case study protocol.** The case study protocol can be described as a workflow:
    1. Keeping track of who refers to who until an appropriate person in an appropriate project to interview is found.
    2. Booking and carrying out the interview. Interview notes are taken.

3. As soon as possible the notes are copied out (sometimes the same day, but in some cases more than two weeks afterwards).
4. The copied out notes are sent to the interviewees for review. This has several purposes: first, to correct anything that was misunderstood. Secondly, to consent to their publication as part of a technical report (considering confidentiality – in several cases the organization do not want to be recognized). Third, in several cases some issues worth elaborating were discovered during the copy-out process, and some direct questions are typically sent along with the actual notes. These thus reviewed, modified and approved notes are used as the basis for further analysis.
5. After some initial analysis, the interviewees (and a few other people from the same organizations, or who have otherwise showed interest in this research) have been invited to a workshop where the preliminary results are presented and discussed, giving an extra stage of feedback from the people with first-hand experience.

- **Develop case study database.** All notes (even scratch notes on papers, notes taken during telephone calls etc.) are kept in a binder for future reference. Also, any documentation is put in the same place. In order to be able to achieve the workflow described above, the stage of the workflow for each (potential) case is informally noted in an Excel sheet (date of last contact, action to be done by whom). All copied out interview notes are stored in a CVS system.

Internal Validity

To ensure accurate descriptions of data (i.e. interview notes), "member checking" was used; this means that all interviewees have reviewed (and edited) the copied out interview notes. Several types of triangulation [96,403] were employed to increase the internal validity: data triangulation involved interviewing several people in the same case and using several types of sources in some cases (documentation and personal experience in addition to the interviews). Observer triangulation was employed in one case, where a fellow researcher participated during the interview, and also reviewed the copied out notes.

External Validity

The main purpose of studying several cases is to achieve a higher degree of external validity. It should arguably be less likely that several cases are included that are extreme in the same way, than when studying a single case. The cases include both larger and smaller organizations, and the software

belongs to several types of domains (see Paper II for details). The cases are theoretically replicated [403] (or analytically generalized [313]) so that there are several indications supporting the same theoretical proposition (e.g. that performing activity *X* is beneficial).

3.3.4 Phase Four: Single Case Study and Formal Model for Merge

One of the cases in phase three was followed up, as it was the case that most clearly implemented the *Merge* strategy. This study was thus a single case study [403], which was carefully selected as an extreme case in a good sense, illustrating how the *Merge* strategy could actually be implemented. Based on the knowledge of the case gather during phase three, interview questions were designed (reprinted in Appendix C) and this time I met personally with all five developers involved in the two systems (distributed on two continents) and made interviews. I was also given high-level documentation of the Swedish system. The case is further described in Paper V, and in more detail (including the copied out interview notes) in a technical report [208]. The rest of this section describes the research method, in particular how threats to the different types of validity were addresses. The text in this section is heavily based on the technical report [208].

Construct Validity

By conducting several interviews, and having some documentation available, there were multiple sources of evidence.

Reliability

Yin's [403] two pieces of advice has been followed as follows:

- **Use case study protocol.** The case study protocol can be described as a workflow:
    1. Formulating research questions and discussion agenda.
    2. Booking and carrying out the interview. Taking interview notes.
    3. Copying out the notes (sometimes between the same day and three weeks afterwards).
    4. Sending the copied out notes to the interviewees for review. This has several purposes: first, to correct anything that was misunderstood. Secondly, to consent to their publication as part of a technical report, considering confidentiality. Third, some issues worth elaborating

may be discovered during the copy-out process, and some direct questions will then typically sent along with the actual notes. These thus reviewed, modified and approved notes are used as the basis for further analysis.

- **Develop case study database.** All notes are kept in a binder for future reference. All copied out interview notes are stored in a CVS system.

Internal Validity

As in phase three, it has been ensured that descriptions of data (i.e. interview notes) are accurate through "member checking", i.e. all interviewees have reviewed (and edited) the copied out interview notes. Several types of triangulation [96,403] were used to increase the internal validity: data triangulation involved interviewing several people in the same case and using several types of sources in some cases (documentation and personal experience in addition to the interviews).

External Validity

The case was the most illustrative and interesting example from a known set of cases (the nine cases of phase three). The other cases of phase three form a sort of background to the selection of this case, and the assumptions and conclusions of the study in phase four are (explicitly) influenced by these other cases. However, there must also be convincing argumentation for external validity. To some extent I believe this phase should be seen as a starting point for future research, preferably by implementing the findings in future cases and evaluate the outcome. In this way, limitations of the current propositions would be found.

3.3.5 Phase Five: Questionnaire Validating and Quantifying Earlier Findings

A questionnaire was designed to collect quantitative data for the previously qualitative observations. By correlating the responses to several questions, the questionnaire would also to some extent validate these previous observations. Therefore, a questionnaire was designed consisting of a number of questions with five-grade scales. The questionnaire was distributed to six of the previous and two additional cases. By returning to the previous cases, some amount of internal validation of our previous interpretations (in terms of theory construction) is ensured. If the respondents would describe the cases in a very different way from what we

have done based on the previous interviews, it is a sign that the theory is a bad representation of the reality. By administering the questionnaire to some cases that were not part of the previous study, we get an indication whether the theory extracted from the previous cases makes sense at all.

The questionnaire form is reprinted in Appendix D and the collected data in Appendix E. The responses were analyzed and published as a conference publication [222] which is included as Paper IV.

Construct Validity and Reliability

There are databases with e.g. all companies registered in Sweden which are typically used to define a population and retrieve a random sample from for similar kinds of surveys. However, the problem in this case is that it is difficult to formulate the population in terms of the information found in these databases. There are no entries for newly merged companies, so one would have to make some assumptions concerning whether company names and organization numbers are identical or change compared to previous years etc. Also, we are interested in the software development activities within a company, which could be found in virtually any business domain, and the size of the company would not necessarily hint at the size or importance of the software department. In addition, we are not only interested in commercial companies, but other types of organizations as well, such as governmental departments or regional official organizations. Although not necessarily impossible, it would require much research and assumptions only to define a population.

Instead, we once again chose to rely on convenience sampling, i.e. calling and talking to people and organizations we considered were likely to have gone through a significant integration effort. We returned to our previous cases as well as pursued some contacts in other organizations. In the previous cases, we tried to get access to more people than we had interviewed before.

The cases and respondents are summarized in Table 1, together with the number of interviews made in our previous study.

Table 1: The Cases and Distribution of Respondents

Case	Organization	System Domain	Number of respondents (previous study)
A	Newly merged international company	Safety-critical systems with embedded software	1 (1)
B	National corporation with many daughter companies	Administration of stock keeping	1 (1)
C	Newly merged international company	Safety-critical systems with embedded software	1 (2)
D	Newly merged international company	Off-line manage-ment of power distribution systems	0 (2)
E1	Cooperation defense research institute and industry	Off-line physics simulation	1 (1)
E2	Different parts of Swedish defense	Off-line physics simulation	1 (1)
F1	Newly merged international company	Managing off-line physics simulations	0 (3)
F2	Newly merged international company	Off-line physics simulation	1 (6)
F3	Newly merged international company	Software issue reporting	0 (1)
F4	Newly merged international company	Off-line physics simulation	1 (0)
G	Newly merged international company	Database-centered system	2* (0)
		Total number of respondents:	**9 (18)**

* The respondents are labeled G_a and G_b in the appendices.

In this kind of study, construct validity can be divided into several requirements: convergent validity means that all items forming the construct should be concurrent, and discriminant validity means that no item should be indicative of more than one construct within in the same setting. When these requirements are fulfilled for all constructs in the same setting, they can be evaluated for nomological validity, which means that they actually reflect the theoretical ideas about the phenomenon.

Internal validity is ensured by our documenting the study in a technical report [221], which includes reporting the questionnaire (also found in Appendix D), the characteristics of the respondents and cases studied (to the extent confidentiality agreements allow), and the responses (in Appendix E).

Internal Validity

Twelve cases were contacted, with a total of around 25 people, to ensure at least one response from most cases. We received responses from eight cases, nine people. The response rate was thus 2/3 of the cases, and ca 1/3 of the potential respondents. Our conclusions per case are therefore sensitive to individual responses, but conclusions where all responses are summed are less sensitive. We expect to continue distributing the questionnaire to more cases and respondents in the future, and the current data should only be seen as preliminary indications.

External Validity

Ideally, the sample size in a quantitative study should be considerable – in this case, hundreds of organizations should be studied. A large sample is needed to give statistical confidence in the findings, and enables statistically significant analyses concerning differences between large and small companies, business domains etc. However, the relative size of a sample needed for a certain confidence level drastically decreases as the size of the population increases.

3.4 Overall External Validity

The research of this thesis can to a large extent be classified as empirical science, but there is an important interpretative element as well (e.g. by providing idealized concepts in order to discuss and explain observations), and also an element of formal reasoning. The research has been mostly qualitative but has also included a phase of quantification of earlier results. It has also included one iteration of validation of earlier results, which means

that they are supported rather than falsified. In each phase, an appropriate research method has been chosen, and the recommendation how to implement it rigorously and systematically has been followed as far as practically possible. The most serious limitation of the result is of a practical nature, in the number of cases studied and the volume of data for some cases (only one interview). With these limitations in mind, a satisfactory amount of validity can be claimed.

Further studies are needed to show to what extent the results are general for a large set of organizations, and to what extent observations and guidelines are dependent on factors such as system domains and sizes of organizations.

Chapter 4. Related Work

This chapter relates the research in this thesis to relevant research and practice, subdivided into three parts. The topic of the thesis – in-house integration – is first related to other types of integration in section 4.1. The existing literature and practice of my two points of view are then surveyed: software architecture in section 4.2 and software processes and people in section 4.3 throughout. The focus is on describing the existing research most relevant for this thesis, and relating this research to the existing research and practice throughout. At the end of each of these three sections, this thesis is explicitly positioned with respect to the related fields.

4.1 Software Evolution and Integration

This section starts with a brief overview of the area of software evolution, and focuses then on various aspects of software integration. This text is based on my earlier surveys of the fields: evolution in [206] and integration in [212].

4.1.1 Fundamentals of Software Evolution and Maintenance

The term "evolution", when applied to software, usually means that a system is modified and released in a sequence of versions [237] (although it is sometimes used e.g. for the self-modification of programs or for evolutionary programming techniques such as genetic algorithms [20,266]). For any software system that is being used, its context evolves: businesses evolve, societies evolve, laws and regulations evolve, the technical environment in which the software executes and is used evolve, and the users' expectations of the software evolve. Therefore, new features and improved quality will be required to keep up with a changing environment and changing user expectations [53,236,237,302]. The term "maintenance"

usually means making relatively small changes, and can be classified into perfective maintenance, corrective maintenance, preventive maintenance, etc. [156,296,305,352]. There is, however, not a clear border between these – or between maintenance and further development – in spite of efforts to define them and create different process models [171]. As these changes accumulate, we call it evolution.

If a system is evolved with tight time schedules, and insufficient time or knowledge of the original design ideas, or insufficient time to revise those fundamental design choices consistently, the conceptual integrity [54] of the system will be violated and deteriorate (or "erode" or "degrade") [24,42,167,299,350,381]. Complexity will increase unless work is done to reduce it [236]. This is a difficult but unavoidable problem: successful systems need to be evolved in order to stay successful, but while being evolved they typically deteriorate and become increasingly difficult for humans to understand and modify further – unless this is proactively managed [380]. The long term optimum should be a proper balance can be found so that maintainability is maintained [205,310], but there are still many challenges to be addressed [264]. When modifying a system it is important to understand the rationale behind a system's design in order to avoid design deterioration [50,232]. Maintenance is not only a post-deployment activity but should be carefully planned already during system's development [174,305]. For anticipated evolution, the architecture can be devised so that certain updates are made easier, in the form of decentralized, post-deployment development of add-ons, scripts, etc. [292]; there are even approaches to update systems in runtime (based on its componentized architecture) [293].

The software organization itself, including its tools and processes, affect how well it performs in maintaining and evolving its software [156,172,191,240,310]. Not only the actual software is subject to maintenance; maintenance could also involve other artifacts such as the software's documentation [3,174,203]. There are process and maturity models addressing maintenance [13,126,161,173]. Certain system characteristics are considered to influence the required support and maintenance efforts (some proposed measures are reviewed below), but these can be understood only in the light of the relation between the system and its stakeholders (both users and maintenance staff) [64,172,191,240].

There is also much research on how various characteristics of the software itself influences how easy it is to maintain (consider e.g. terms such as maintainability, modifiability, extendability, flexibility, and portability [24,31,154,408]). Large size and high complexity are often considered

making a program difficult to understand, and consequently difficult to modify. There is not a single definition or measure agreed upon, although many measures have been proposed. These include source code measures (including Lines Of Code (LOC), number of statements, numbers of commented lines, and control structure measures [3,14,75,224,285,286,389,408], and various complexity metrics [3,131,256,286,337,338]. The Maintainability Index (MI) aggregates several of these measures into one [286,339]. There are also proposed measures at the architectural level, such as variants on number of calls into and number of calls from a component ("fan-in" and "fan-out") [30,108,124,140,165,224], These are static measures; tracking changes in these between versions could be a way to monitor software deterioration [14,75,205,234,235,311,312,370].

Software evolution results partly from small changes (maintenance) being accumulated, but also from more drastic changes made at various stages in a system's life [237]. For example, web-enabling a system [168], moving from batch execution to real-time service [223] and/or componentizing an existing system [142,168,262] represent such major leaps – as do in-house integration.

There is literature how evolution has been addressed at the architectural level [66,169,260], not least in the form of architecture level evaluation methods based on change scenarios [24,31,70,181,195,247] (see also section 4.2.5 on architectural evaluation). In this context, the field of reengineering can also be mentioned, which includes e.g. how to extract architectural structure from source code [25,46,46,62,127,327,370].

The rest of section 4.1 concerns the particular types of evolution that is related to integration.

4.1.2 Software Integration

The IEEE Standard Glossary of Software Engineering Terminology [154] defines *integration* as "the process of combining software components, hardware components, or both, into an overall system". The fundamental concepts of interfaces, architecture, and information and their relation to integration are first briefly described, followed by surveys over existing fields of research which in one way or another involves integration of existing software.

Interoperability is the ability for two or more software systems or components to communicate and cooperate with one another despite differences in language, interface, and execution platform [388,393]. To be able to do this, components need to have the same understanding of their *interface*, i.e. the "shared boundary across which information is passed" [154] or " a point at which independent systems or components meet and act or communicate with each other" [123,322]. An interface in this wide sense in practice includes such different technical solutions as function signatures, shared memory and variables, protocols for transactions, and file formats.

When two software systems or components are to be integrated, there is a risk that their understanding of the shared interface is incorrect. For example, there is a problem if two components each assume they control the overall execution of the system and will call other components upon demand. This "architectural mismatch" as it has been called [112,115] will result in system malfunction, or no possibility to integrate at all. Architectural mismatch has been noted not only when complementary components are to be integrated, but also when two object-oriented frameworks [44,106,107] – structures assuming to be in charge of the high-level design decisions – are used simultaneously [253]. A survey over the field of architectural mismatch gives by hand the research is relatively immature [36]: there is more research to be found on how to detect mismatches [32,89,90,102,400] than how to solve them, and these approaches are typically not validated in a commercial, large-scale industrial environment. Some interaction mechanisms could be deferred until integration [93]. Some design patterns [113] and architectural patterns [58,110,189,329] address reuse, maintenance and evolution, but there is to my knowledge only little research on design patterns facilitating integration [186,244,402].

A standard architecture into which other components and systems can be plugged may be a viable integration solution within a specific domain. This requires a vendor strong enough to develop an implementation of the architecture, and successfully market it, such as ABB with its Industrial IT architecture [49]. Without a strong vendor, architectures may still be the integration enabler by means of a standard *reference architecture* (reached through common consensus) [265,343]. Component models such as CORBA, COM, J2EE, Koala and .NET – some of which are proprietary and some are defined by common consensus – embed architectural decisions and may be considered middleware architectures facilitating interoperability [404].

To be able to integrate systems, the systems' views of the data – i.e. their *data models*, *taxonomies*, or *ontologies* [128] must also be integrated, an

undertaking not so trivial [152,206,287,306,358]. Geographic Information Systems (GIS) [57,243] is one significant example of a domain where ontology integration has attracted attention [72,283,345,385]. Closely related is the integration of databases, i.e. repositories implementing ontologies. However, the literature to be found is typically fairly old; the problem is today considered part of the Enterprise Application Integration (EAI) approach presented in section 4.1.5.

4.1.3 Component-Based Software

In the field of Component-Based Software Engineering [19,80,138,323,362] software components are viewed as black boxes with contractually specified interfaces. By building a system from pre-existing components, systems could be built faster and cheaper, with the same or higher quality of a system built in-house [83,138,362]. There is a strong focus on explicit interfaces [19,83,138,265,342,362,387], which – to enable true interoperability – must be specified and implemented according to common rules. Component models are such sets of rules, supported and enforced by component technologies, such as CORBA Component Model [351] (a standard from OMG [288] with several vendor-specific implementations), Java 2 Enterprise Edition (J2EE) [269,314] (originating from SUN Microsystems), and COM [47] and .NET [365] (from Microsoft).

The existence of such common integration rules has provided the foundation for a component market where components are developed and used by different organizations, so called “off-the-shelf” (OTS) or “commercial-off-the-shelf” (COTS) components [265,387]. Even when a system is completely developed in-house, a component-based approach may be chosen: a product line approach [69] means that there is a strategy for internal development of components to be reused in a familiy of products with certain similarities. This approach poses new challenges to the software community, e.g. mechanisms for variability to enable evolution of the products of the product line [361], new and stronger mechanisms to track changes to prevent the common assets from degradation [167,360], configuration management to control product derivation and evolution at the same time [375,376], and how to use stakeholder scenarios to evaluate the suitability of a product line architecture [195]. It is of course also quite possible to develop a single system completely in-house according to the component-based paradigm and utilizing an existing component model [45] (this is further discussed in section 4.2.2).

The current component models make integration at the function call level relatively straightforward, but Interface Definition Languages (IDLs) can typically only achieve syntactic interoperability [175,394], which is not enough to make two components interact as desired [388,404]. To ensure true interoperability between systems or components, the semantics must be specified as well [137,249,281,388].

There are some challenges left for the component-based research community. If a component is updated, this may have unpredicted system effects, often due to subtle semantic differences, which calls for new types of evolution support and configuration management techniques [225,227,375-377]. Using third party components in a long-lived system creates an undesired dependency regarding maintenance, updates, error corrections, etc. There is not yet a standardized way of certifying component quality and behavior although there is research on how it could be achieved [83,143,226].

4.1.4 Standard Interfaces and Open Systems

An *open system* is defined as a set of components with interface specifications fully defined, available to the public, maintained according to group consensus, and in which the implementations of components are conformant to the specification [265,342]. Anyone may produce (and profit from) implementations of that specification. (There are also other notions of "openness", less relevant for this thesis, e.g. focusing on mechanisms that allow for third-party extensions of the system [272].)

The notion of open standards is widespread. Major organizations for software standards are ANSI [12], IEEE [157], and ISO [160]. Open systems with standard interfaces (in the form of protocols) are prevalent in computer networks and telecommunications, where customers' requirements on interoperability between vendors is one of the major driving forces [130,278]. Other fields in similar contexts, where systems from different vendors need to interoperate and exchange information are Geographic Information Systems (GIS) [56,72,187,231,283,364,371] and hypermedia [10,11,88,152,267,390] to mention a few. Application domains with an identified need to create their own standard interfaces for interoperability include – just to illustrate the applicability of the approach – public libraries [294], mathematical computations [230], and photo archives [197]. Interoperability through standardized interfaces is also a concern of software agents [231,396]. Although autonomous, agents need to communicate and

exchange data, and to enable interoperability between agents developed with different technologies this needs to be done in a uniform manner [231].

XML [133,399] has become a popular encoding language which may be a common denominator of systems and used for integration [11,65,92,399]. From an integration point of view, the importance of standards applies not only to interfaces but domain-specific architectures as well (see discussion in section 4.1.2).

It is by definition impossible to demonstrate interoperability capabilities in isolation, i.e. without specifying something concrete a component should interoperate with. Conformance testing is carried out to show conformance to a standard, while interoperability testing means testing whether two products (said to adhere to the same standard) actually work together as intended [188]. Conformance to a standard is in practice not enough to ensure interoperability between two implementations [55,238,255].

There appears to be two major reasons for building systems based on standard interfaces. First, building open systems is suitable when an integrator wants to avoid being dependent on a single vendor [104,315]. Second, when there is no single integrator, the only possibility to make different components and systems interoperate is to ensure they conform to a standardized interface [294].

To have a practical impact, standards need implementations. A drawback (from the interoperability point of view) with standards is the commercial marketplace itself with the option for implementers to adhere to standards or not – the choice depends on commercial forces. Another drawback is that reaching consensus often takes a long time, and both vendors and acquirers may need to act quickly in order to produce products and integration solutions on time [239]. This may lead to a number of similar but incompatible de facto-standards. Also, a vendor strong enough may provide an implementation violating the standard and force its competitors to follow.

4.1.5 Enterprise Application Integration (EAI)

Enterprise Applications are systems used to support an enterprise processes (e.g. development, production, management), such as *Enterprise Resource Planning* (ERP) ERP systems [48,84,233,282] systems such as SAP R/3 [324], *Product Data Management* (PDM) and *Software Configuration Management* (SCM) systems [81], and electronic business systems e.g. for business to business relationships, B2B [242,401]. As enterprises need to

streamline their processes to be competitive there is a need for integrating these systems [134,229] to make information consistent and easily accessible. The typical solution is "loose" integration, where the system components operate independently of each other and continue to store data in their own repository [129]. Since building unique interfaces between each pair of systems that needs to communicate is not cost efficient [104], numerous systematic approaches are used to enable a more structured integration of enterprise applications [134,233,244,358]. These are collectively called *Enterprise Application Integration* (EAI) [84,169,241,242,317] and include activities such as new types of requirements engineering [368], data mining and reverse engineering [5] and content integration [358] (to understand the existing data and systems), migration [52] (to get rid of the most problematic technologies and solutions), using a common messaging middleware [51,241,242,258,317,404] (of which there are many commercial solutions), and encapsulating and wrapping legacy systems in a component-based approach [328]. The market for application integration and middleware (AIM) is estimated to $6.4 billion worldwide in 2005 and is expected to continue growing [79].

EAI requires a high degree of commitment, coordination, and upfront investments [233]. EAI may break down when integration occurs between enterprises, when data is operational rather than historical, and more unstructured data need to be integrated [358]. And the integration problem continues: although systems to be integrated may use commercial technologies supposed to support integration and interoperability, these integration-enabling technologies are often not fully compatible, and therefore the need arises to integrate these integration technologies [122].

4.1.6 Product Integration

Product integration is the part of systems engineering when the individually developed parts of a system is assembled into a whole [73,103,228,322]. In this context, the system is often designed top-down, after which follows the implementation of the various parts (and possible acquisition of existing components), which are then integrated [103]. The integration activity should not only be carried out solely in the end, but should preferably be carried out throughout development; there is e.g. the practice of building the product daily, performed in order to get early indication of integration problems [257,284]; this will also give a hint of the emergent system

properties. Interface specification and coordination are important activities [73,103,123], and a systematic implementation of the component-based paradigm enables parallel development and (hopefully) smooth integration [228].

4.1.7 Merge of Development Artifacts

As development artifacts are branched and developed in parallel, there is a need to merge them – this is typically an integral part of a software configuration management system [34,263]. There are many methods and algorithms for doing this, the simplest kind of which is a textual comparison and merge; these are generally applicable but can only resolve very basic conflicts however [263]. By narrowing down the application domain to e.g. a specific programming language, it becomes possible to perform a syntactic merge [263]; also resolving semantic conflicts is more difficult, and it is in general an undecidable problem [33,372]. The parallel development branches may be refactored differently, which gives rise to structural conflicts which need to be resolved in order to enable a merger at this higher level; this however seem to be a largely open research area [263,379].

In practice, the larger the difference between two development branches, much more user feedback and coordination is needed to resolve the conflicts than if the same changes are being made but the conflicts are frequently resolved and the branches merged [15,263,304]. (This means, extrapolated for the in-house integration context, that when merging two systems that have evolved independently for many years – and perhaps had nothing in common to start with – the available merge algorithms would be essentially useless.)

4.1.8 Positioning this Thesis in the Context of Software Evolution and Integration

Software evolution is a consequence of changes being accumulated, both small and large; in-house integration represent a major change in direction for a system. Existing research on integration usually considers integration of components complementing each other (rather than creating one entity out of two, as is the case in in-house integration). An important rationalization goal for in-house integration is reduction of the systems to be maintained and supported, while the surveyed fields aim at acquiring and integrating

external components or systems. Also, the organizational context is often different from in-house integration: typically, these existing fields assume that the systems or components are developed independently by third parties – the open systems approach even assume there is no single integrator – while in-house integration concerns existing systems completely controlled in-house. The processes are also different compared to in-house integration: product integration, and to some extent component-based development, involves a top-down design process. Nevertheless, these fields are applicable to the in-house integration context to some extent: viewing the internal structure of components of a system – possibly also componentizing them first – aids their integration. The existence of standards is also of benefit to in-house integration (if the standards are adhered to).

Integration in the sense of "creating a single entity out of two (or more) existing pieces of software" we have found only in two senses: first, it is discussed for development artifacts (most often source code files). One prerequisite for any useful merge would be that the systems are written in the same programming language – possibly it would be possible to first convert the source code of one system into a functional equivalent in another language [366]. These approaches however assume many similarities between the two artifacts – their original purpose is to enable merging branches of the same artifact. Even in the two studied cases where the systems have been branched from a common ancestor, they have diverged for many years. Also, the unit of reuse when merging source code files is statements or lines of code, which is a far too low abstraction level to be applied to large complex systems. Finally, the major challenge during in-house integration is not technical, as it involves complex requirements, functionality, quality, and stakeholder interests. Second, we have seen the observation that the usage of two object-oriented frameworks simultaneously can be expected to bring problems due to conflicting assumptions [253].

Other observations of architectural mismatch are clearly applicable during in-house integration, although there is little or nothing to be found that directly can help detecting and solving the mismatches [36]. This research complements existing reports by identifying particular incompatibility problems found during in-house integration.

Numerous other surveys of software integration have been published previously [122,137,169,233,263,295,355,388], each done from a particular point of view – ours of course with the purpose of investigating the extent to which in-house integration and this research is unique.

4.2 Software Architecture

This section provides a broad survey over the area of software architecture, one of the approaches adopted in this thesis to address the in-house integration challenge. Parts of this section have been adopted from [206].

Today's notion of software architecture can be traced to early suggestions that the need for humans to understand a system should guide its decomposition rather than considerations on e.g. performance [54,99,297]. Object-oriented analysis and design partly addressed this as systems grew larger [39,163,318]. The foundations of the field of software architecture were laid [1,95,117,303,349] as the first architectural languages were designed [245], the need for views [199], the rise of the pattern community [58,113], special issues of journals [155], and the first books [58,350].

4.2.1 Definitions and Use of Software Architecture

Academic research has focused on software architecture in the sense "structure of components". This can be seen in definitions of architecture [25,58,159,350], in the notion of *Architecture Description Languages* (ADLs) and *views* (see section 4.2.3), and *patterns* or *styles* (see section 4.2.4). When viewed as a design tool, this paradigm focusing on structure is valuable as it raises the abstraction level and discusses the connections between components explicitly [346]. However, this view is limited; there is for example limited value in visualizing code structure without knowing the intentions behind the design [25,40,46,62,127,327,370]. Some texts emphasize architecture as a set of design decisions [166,378,397], an important part of which is their rationale [71,159,303] which is important for maintainers will arguably be able to perform changes efficient and without violating the conceptual integrity of the system [50,232] (cf. the discussion in software deterioration in section 4.1.1). Some texts talk about architectural knowledge [201], and thus turn the notion to architecture being a social construct, describing architecture as a concise explanation of whatever is important about a system, or whatever about a system that must be understood by all developers (possibly agreed upon through group consensus) [111,200]. Different stakeholders have different needs of an architecture (and its description) that should be addressed [25,71,159], and the business and organizational context of a system could also be considered an essential part of the architecture [289,405,409]. Along this line, the role of the *architect* is perhaps as important to discuss as the architecture

[100,151,193,252,273,344], and there are associations of architects [153,398].

Nevertheless, it is in the sense of structure that architecture has been most researched – possibly because it has proven relatively easy to formalize. However, it is no longer only tied to the early design phase but plays an important role during the complete life cycle of a system [25,43,66,200,301]. In this thesis, this trend is continued by describing the central role the systems' architectures plays during in-house integration – not least its representation and documentation.

An architectural description serves as a communication tool between stakeholders of the system, so that the e.g. managers, customers, and users understand a system's possibilities – and limitations – in areas of their concern [25,70,71,159]. Architectural descriptions can also be analyzed, which makes it possible to evaluate alternative architectures before a system is built [25,68,70,180,182,183] (see section 4.2.5).

There is a correlation between the structure of an organization and that of its software [77,134]. The integration may occur at different levels, ranging from data and application to the more difficult levels: business processes and humans [307]. The "Zachman Framework for Enterprise Architecture" is a framework within which a whole enterprise is modeled in two dimensions: the first describing its data, its people, its functions, its network, and more, and the other dimension specifying views of different detail [405,409]. Another enterprise information systems framework is "The Open Group Architectural Framework" (TOGAF) [289].

4.2.2 Component-Based Architectures

As described in section 4.1.3, the component-based systems paradigm may be adopted for systems built completely in-house (i.e. even if third-party components are used, and no product line is built). By using a component model, the architecture has to be explicit, interfaces have to be explicit, interactions are explicit, and the architecture may be loosely coupled [45]. Choosing a component model in effect means that certain architectural choices are also made, since the different component models are designed for different types of systems with different requirements [45]: CORBA [351] (a standard from OMG [288] with several vendor-specific implementations) is designed for distributed real-time and performance-critical applications, Java 2 Enterprise Edition (J2EE) [269,314] for distributed enterprise systems, and COM [47] and .NET [365] address the

desktop domain. These component models have been compared from this architecting point of view [97,105,404].

For any system, it is very difficult to predict system properties from component properties, since the system properties are affected not only by the components themselves but also by their configuration and interaction. However, thanks to the restrictions posed by a component model system properties could possibly be aggregated from component properties – if there is enough information about the components, which for components developed for the marketplace would require some certification system [143-147,226,270,354].

4.2.3 Views and Architecture Description Languages

An important aspect of a software system's architecture is, as said above, its structure. However, depending on the point of view, it is possible to discern not only one structure but several, superimposed one upon another [54,298]. This has led to the notion of *views*, i.e. a "representation of a whole system from the perspective of a related set of concerns" [159]. When discussing systems in general, the appropriate term to use is *viewpoint* [159] or *viewtype* [71], which refers to the perspective itself rather than a particular system's representation.

There are some viewpoints that seem to be almost universally useful, such as those of the "4+1 views" where a logical view, a process view, a physical view, and a development view are complemented and interconnected with a use case view [199]. There are slightly varying names of essentially the same viewpoints, such as the suggested conceptual view, execution view, module view, and code view [58,71,149]. There are also suggestions of additional views that could be useful in some cases, such as an architectonic viewpoint [250] and a build-time view [369]. There is research on how to formally relate elements of different viewtypes [139,271,407]. As a documentation of a system, an architectural view should not only contain the actual structural description but also specify which stakeholders and concerns it addresses, and the rationale for using it [159].

Visual representations are intuitively appealing to humans, and with well specified syntax and semantics, such an *Architectural Description Language* (ADL) also enables formal analysis, and possibly translation into source code. The *Rapide* language is both an architecture description language and an executable programming or simulation language [245]. The Carnegie Mellon University has constructed several ADLs as part of their research,

such as *UniCon* [350], *Aesop* [114], and *Wright* [7]. The research community have produced many other ADLs with exotic names such as *ArTek*, *C2*, *CODE*, *ControlH*, *Demeter*, *FR*, *Gestalt*, *LILEAnna*, *MetaH*, *Modechart*, *RESOLVE*, *SADL*, and *Weaves*; see e.g. [261,333,336] for further references. *Acme*, developed by a team at Carnegie Mellon University is designed to be an interchange format between other languages and tools [116], but should possibly be considered as a new ADL in its own right [87]. The *Architecture Description Markup Language* (ADML) is an XML representation of Acme with some extensions and transparent extensibility [291].

Koala is an ADL and component model used at Philips to develop consumer electronics products such as televisions, video recorders, and CD and DVD players [382,383]. The *Fundamental Modeling Concepts* (FMC) [136,184,185] focuses on human comprehension and supports representations with three different views: compositional structures, dynamic structures (behavior), and value structures (data). FMC has successfully been applied to real-life systems in practice at SAP, Siemens, Alcatel and other companies, and has also been used in a research project to examine, model, and document the Apache web server [127,135].

The *Unified Modeling Language* (UML) originated from object-oriented design and modeling [41,149,373], but is also used for modeling non-object-oriented software as well as for systems engineering. Although UML met some criticism from the architectural community [74] it became the de facto language used in industry to model architectures [194,198,261]. UML 2.0 [290] provides more capabilities for modeling architectures, and it provides several extension mechanisms that may be used to support architectural constructs [41,259,319]. It could still be confusing to use the same notation for different levels of abstraction [148].

4.2.4 Styles and Patterns

An *architectural pattern* (or *style*, or *design pattern*) is an observed, recurring way of solving similar problems, which are proven to have certain general properties [24]. There are generally applicable patterns [58,113] as well as patterns for various domains, such as distributed systems [329], resource management [189], and enterprise systems [110]. Attempts have been made to formalize what constitutes a pattern in a formal language [1], but so far the great impact of patterns has been at the level of increasing the knowledge of developers and architects. There are even more ambitious projects that aim at systematically collecting experiences and patterns from

successful software systems [40]. As with views, styles abstract away certain elements and emphasize others [71], and it is often appropriate to describe the same system with several styles simultaneously [25].

There are some styles commonly found in literature. Systems where data flow is in focus may be described with the *pipe-and-filter* style [24,71,349,350,386]; a simple compiler is typically considered a typical example of a pipe-and-filter architecture [4,350]. A *blackboard* (or *repository*) architecture draws the attention to the data in the system [24,349,350,386]. In a *client-server* architecture [24,42,341,349,350,386], the system is organized as a number of clients (typically not aware of each other) issuing requests to a server, which acts and responds accordingly. With a *layered* architecture, focus is laid on the different abstraction levels in a system, such as the software in a personal computer [24,42,349,350,386]. An *n-tier* architecture is typically used for business and information systems and illustrated as a database at the bottom, a user interface at the top, and possibly a separate component executing the business logic [25,71,340]. Object-orientation has also been discussed as an architectural style [24,42,350], but perhaps it is rather a high-level paradigm than a style. The control loop paradigm for control systems has also been characterized as an architectural style [350,386]. There are many variants of these styles differing with regard to e.g. implicit or explicit invocation, and there are also more styles and patterns listed in literature [58,110,113,189,329].

4.2.5 Architectural Evaluation and Analysis

Before a system is built (or before implementing some changes), one would like to predict certain properties in advance. If no system yet exists, one need to analyze architectural models; preferably several alternatives are evaluated [25,68,70,180,182,183]. Given a description in a formal ADL it is possible to analyze it statically for consistency and completeness with respect to some property of interest [7,98,101], or consistent implementation of a style [1], or to execute or simulate it to assess other properties [6,7,21,22,245].

There are techniques to evaluate system performance based on architectural descriptions [98,101]. Certain qualities may also be assessed through simulation of an architectural description [245] or prototyping [21,22]. Formal techniques in which entities are grouped into clusters based on a similarity measure can be used to analyze software structures and improve various architectural concerns such as hardware parallelism (if clusters

represent nodes) or ease of change (if clusters represent source code modules) [25,246,268,330,391].

Another type of analyses should be made by involving the system stakeholders [24,41,42,182,183,196,316]. The *Software Architecture Analysis Method* (SAAM) uses stakeholder-generated scenarios to compare the quality properties of alternative architecture designs [24,70,179,180]. The *Architecture Tradeoff Analysis Method* (ATAM) is a successor of SAAM which focuses more on business goals and making tradeoff points in the architecture design explicit [70,70,183]. ATAM and the *Good Enough Architectural Requirements Process* (GEAR) address the practical need for tradeoffs and "good enough" solutions [331]. The *Active Reviews for Intermediate Designs* method (ARID) is appropriate at an earlier stage than ATAM, as it involves stakeholders to evaluate partial architectural descriptions, also with the help of scenarios [70]. The *quality attribute-oriented software architecture* design method (QASAR) puts architectural analysis and evaluation in an iterative development context, where a design that fulfills functional requirements is refined until quality attributes are satisfactory [42]. *Architecture-Level Modifiability Analysis* (ALMA) method focuses on assessing modification efforts of an architecture, also based on change scenarios [31]. The *Cost-Benefit Analysis Method* (CBAM) focuses (as the name suggests) on the trade-offs between costs and benefits which must often be made early and will be embedded in the architecture [70].

Most of these analysis methods serve as frameworks leading the analyst to focus on the right questions at the right time, and almost any imaginable quality attribute could be analyzed; there are e.g. experience reports from evaluations of modifiability, cost, availability, and performance [70,181-183,204]. Apart from its direct results in terms of analysis results, these kinds of informal analysis also increases the participants' understanding of the architecture and the trade-offs underlying it [24,182,183].

4.2.6 Positioning this Thesis in the Context of Software Architecture

For in-house integration, architecture is considered to be anything that is relevant to discuss about the systems, which could otherwise cause incompatibility problems during implementation, and which can be discussed at a high enough level. This includes not only structure, but the data models has also been identified to be a potential cause of problems, as has also what this thesis has been labeled "framework" (in a different

meaning than object oriented frameworks [170] and component based frameworks [83]).

The need for representing the architectures of the existing systems has been emphasized. This could be done using any language that is feasible, meaning easy to learn and simple to use – to enable rapid construction of alternatives and their evaluation – while powerful enough to capture the issues of importance for the specific case. I have largely avoided suggesting any particular language or view, and I am currently looking for alternatives to the module viewpoint currently used by the method and tool for exploring *Merge* alternatives.

The notion of patterns and styles shows how strong the research focus so far has been on structure (patterns and styles are restrictions on structure). The conceptual integrity of a system was briefly assessed in the case study of research phase one, and includes the existence of simple, well-known, consistently implemented patterns. In this research, indications where that two systems often have structures with more similarities than can be explained by pure chance; possibly, patterns and styles could be an additional explanation.

For in-house integration, as many alternatives of future systems as is practically feasible should be developed, and these should be evaluated as thoroughly as is practically feasible. This could possibly be done by using elements of the established methods reviewed above. Evaluating architectural compatibility however is unique to the context of integrating several systems, and there is unfortunately little published knowledge to apply [36] (cf. the discussion in section 4.1.2). Our studies have shown the usefulness of the simple evaluation method of associating implementation and modification effort to individual components, which enables a continuous evaluation during exploration of merge alternatives.

4.3 Processes and People

The topics covered so far – evolution, integration and architecture – are inseparably bound to a process context. The organization has some overall goals of the integration which must be achieved in reasonable time, by people with different skills and roles. This section will therefore survey the field of software processes, and other closely related issues will also be touched upon, such as the role of an organization and its people.

For the purpose of presenting this overview, the word "process" is used for the activities performed in an actual project – this is what is possible to observe in a study. The term process is used both for the overall process (as in "development process") and sub-processes (such as a documentation process, an integration process, etc.). Generalized descriptions are called "process models", involving abstract activities and stakeholder roles that need to be customized for a particular project and instantiated as concrete processes [118,119,162,251,309,352,384]. A brief overview of these is provided in section 4.3.1, including higher-level models such as maturity models and process standards. Section 4.3.2 focuses on concrete practices that are part of many process models at various levels.

4.3.1 Process Models

The earliest and most basic development model is the sequential *waterfall model*, according to which there is a strict sequence of development phases: requirements for the system are first gathered, followed by design and implementation, integration, testing, and deployment [309,352,384]. With a proper division into separate system parts, it is possible to develop these parts in parallel and thus shorten the total development time needed [309], and conversely: a well modularized architecture may be a requirement e.g. when development teams are geographically distributed [60,178]. There is little room for evolution in sequential models; errors introduced in one phase may easily remain undiscovered until its corresponding verification and validation phase (often illustrated in a "V"-shaped diagram) [119,251,309,352,384] – the earlier the error is introduced, the later it is discovered. Integration is performed in a phase towards the end of development, when all individual components are assembled into a system, and is followed by a system test phase; not until this phase is it possible to observe system properties. By prototyping, and/or performing development in iterations or increments, the system is being built evolutionary, which makes it easier to monitor progress, increase customer feedback, and evolve the requirements and implementation along the way [26,28,37,257,357,384]. With short enough iterations, or small enough increments, early feedback is provided as of potential integration problems as well as system properties [228,257].

There are processes with strong roots in a particular technology, most notably perhaps being object-oriented development [39,163,318]. This has influenced the Unified Process (UP, further evolved by Rational into the

Rational Unified Process, RUP), which utilizes the UML language and focuses on iterations and incremental development [119,200]. UP and the agile movement to a large extent build on good practices [26,28], which are further discussed in section 4.3.2. As industry is turning towards component-based development, the previously prevalent top-down approach must be complemented with a bottom-up survey and assessment of existing components [18,19,35,36,80,82,126,164,176,275]. There may be existing components to reuse, developed in-house or by some external party, but they cannot be expected to perfectly match the requirements; this, together with the difficulty of finding (reliable) information about commercial and open source software components [17,36,86], means that the development process must be radically different from the top-down approach, and involve e.g. prototyping using potential components [35,36,164]. The recent trend of rapid development and agile processes emphasize the need for customer involvement and continuously adapting to new requirements [26-28,257,357] (and also, we can note in the context of this thesis, continuous integration of work products [257]).

There are models and standards at a higher level, in the sense that they describe not concrete activities that make up a process, but rather define terminology and process areas; they typically also discuss the complete software life cycle process – not only the development process [73,103,161,162]. An overall goal of these models is to make projects and their outcomes predictable; actually terms such as "software production" and "software factory" has been used instead of "development" [119,384]. Typically, these standards and models define different process areas and their goals, and then suggest the implementation of specific practices considered to achieve certain goals (these practices are further discussed in section 4.3.2). The practices implemented in an organization signify its maturity level, ranging from *incomplete* (only in the CMMI), *performed*, *managed*, *defined*, *quantitatively managed*, and *optimizing* [73,161].

When the same software process is repeated – as in a manufacturing process – it becomes possible to analyze it and improve it. Process improvement can also be viewed as a process, and as such it has its own process models, such as the *Initiating, Diagnosing, Establishing, Acting & Learning* (IDEAL) Model [125,334]. The *Six Sigma* approach originated in hardware manufacturing, which focuses on continuous measurement and improvement of defect rates, is also being applied [335]. The reviewed process models and standards are typically used to guide such process improvement initiatives, e.g. by moving up maturity levels [13,73,161,173].

These higher-level process models can be criticized for being described at a too high level while also being inflexible, making them difficult to implement properly in an organization [59,279]. One can also argue that these models treat people as parts of a production machine, which suffocates creativity within an organization [94]. And true, standards and defined processes intend to raise the skill level from individuals to the organizational level – but they can never be a substitute for skilled people or proper understanding of why things are being done. Rather the opposite: a proper implementation of the EIA-731.1 is explicitly said to include "skilled personnel … to accomplish the purpose of this Standard" [103].

4.3.2 Good Practices

The available literature suggests many individual practices that are known to minimize project risk or improve performance. Many of these practices can be employed somewhat independently, but they also interact.

It is important to provide a productive environment where people feel comfortable, free from interruptions and background noise [94,257]. Learning should be promoted and the staff's skills continuously improved [85,103]. It is essential to provide a constructive atmosphere by focusing on common interests, creating win-win solutions, and agreeing on objective criteria rather than protecting positions [257].

To become productive, stakeholder commitment to a project is essential for success [2,73,126]. To achieve this, people need motivation, which may be achieved by defining intermediate goals – if they are perceived as realistic and come at reasonable frequency [257,352,357].

Proper planning involves planning project resources, defining the required knowledge and skills, assigning team members with appropriate experience and skills, defining roles and responsibilities and ensuring involvement by the right stakeholders at the right time [73,251,309,352,384]. Defined goals, planning and monitoring are essential for project success, preferably with quantified goals and metrics [73,120]. Project risks must be analyzed and addressed, early in a project and throughout [73,103,118,309,352].

Whatever work products are produced, verification and validation of these are a strongly advocated means of quality assurance [73,103,119,251,309,352,384]. Although automated tools should be used in testing and analysis of work products (e.g. source code), peer reviews and

inspections are essential to increase quality and thus reduce costs in the long term [73,118,132,257,352].

The agile movement seem to advocate a less formal requirements engineering, and instead focus on continuous customer involvement [26-28,357]. This is partly explained by their typical different application domains and project sizes [38]: the more heavy-weight processes address large-scale, critical software [73,162], agile methods seem to be most suited for small-scale projects of non-critical software. Agile methods embrace change, and continuous redesign and refactoring of the architecture is an important activity (as opposed to considering architecture design as a separate early phase) [26-28,109]. It has been suggested that a system's architecture plays an important role in all life cycle phases of a system [25,43,200,301].

As organizations become global, they face the challenge of employing distributed processes, i.e. where the team members are geographically dispersed [60,141,178]. There must be technical infrastructures in place that support collaboration over distance, and there are cultural differences that need to be understood and properly managed [60,61,126,150]. Meeting in person regularly is essential, and more formalized legal agreements are advised [60].

4.3.3 Positioning this Thesis in the Context of Software Processes

In-house integration presents many process challenges, as reported in Chapter 2. In this thesis I have chosen to not create a very comprehensive model, but rather point out some high-level issues to consider and some low-level practices. This should make the results relatively easy to implement with little impact on existing process in an organization. An additional challenge, inherent in the in-house integration context, is that the newly merged or closely cooperating organizations will also need to integrate their two different sets of processes, and I do not want to add additional complexities to this already challenging endeavor.

As the focus has been on the vision process, aimed at outlining the requirements and architecture for an integrated system, a need for something in line with the agile movement has been observed, i.e. a rapid, high-level discussion where it is essential to provide a constructive working atmosphere. The implementation phase of in-house integration could and

should follow any process model the organization is familiar with, while also considering issues pointed out by this research, such as the need for stepwise deliveries due to the long time scale of many integration alternatives.

Many suggested practices found in literature were found to be beneficial also in the context of in-house integration, such as the importance of assigning the right people and provide constructive atmosphere, and achieve stakeholder commitment. The vision process is aimed at planning the implementation process, and the known good practices of project planning apply here, such as planning resources and schedules, identifying risks, and ensuring stakeholder participation. The vision process itself also needs to be planned, and here the need to plan for using the most skilled and knowledgeable people has been emphasized. Concerning the system requirements, the presented findings are somewhat different from many suggested practices found in literature: since requirements engineering has already been done for the existing systems, it is possible to let users and other stakeholders evaluate these systems; in this way they will both formulate requirements and evaluate the existing implementations of these requirements. A light-weight requirements gathering phase is therefore advocated, where requirements may simply refer to the existing systems. An organization embarking on in-house integration is by nature distributed, which requires awareness of the challenges and practices involved with distributed teams, often involving different cultures.

The in-house integration process is by nature a process repeated very seldom within an organization, and it makes little sense to talk about process improvement. This makes the research of this thesis even more valuable, as experiences are collected from multiple organizations that help organizations designing a feasible process and avoid some pitfalls the first time.

Chapter 5. Conclusions and Future Work

The goal of this research has been to develop a systematic approach that will lead to a more efficient and predictable process for executing future in-house system integration projects. To achieve this, we have surveyed, evaluated, and generalized existing practices in a number of organizations, which has resulted in the formulation of guidelines for the in-house integration process. The thesis focuses on the early vision process, which should be carried out relatively rapidly while ensuring enough coverage of the most important considerations.

One such important consideration is the architectures and compatibility of the existing systems. If the systems are not sufficiently similar with respect to structure, data models, and frameworks (including the technologies used), it is not possible, in practice, to combine their components. Surprisingly, similarities are more common then what could be expected, however, and some indicators suggest a certain amount of compatibility: systems more or less contemporaneous often exhibit similarities, and certain solutions are often implied and required by domain standards. For many other considerations, there is no objective way forward to the integrated system; rather, it is most important to involve the right stakeholders at the right time (for which we provide some guidelines) while ensuring a balance between collecting opinions and performing a detailed analysis on one hand, and rapidly making a decision on the other. What makes the decision-making of in-house integration exceptional is that the stakeholders initially know their own system well, but know little of the other existing systems. The existing systems must be evaluated and alternative integration solutions must be created in a neutral way, and this thesis contributes to this by presenting practices of decision-making, evaluation of the existing systems, and creation of alternatives of a future system that ensure both efficiency and objectivity. This is difficult, however, because decisions that may be made are not necessarily in the interest of all persons involved in an in-house integration.

The thesis provides guidelines at a very high level, as well as at a very practical level. These guidelines are therefore of value to those integrating

in-house software systems because they are compatible with most processes used in organizations, whether they are plan-driven, agile, or ad-hoc. In this connection, this thesis is particularly focused on the significant characteristics of the decision-making phase of in-house integration. The implementation phase will be more similar to processes with which organizations are more familiar: development of a new system (or system parts), evolution of systems towards a desired state, and retirement of systems; however, there are several challenges specific to in-house integration in this phase as well. Several processes will be mostly run in parallel, and they must be properly synchronized and managed. This becomes more important the tighter the proposed integration solution is. If an evolutionary merge is attempted, i.e. the existing systems are evolved in parallel and released separately until the merge is completed, there must be short-term benefits also for the existing systems. In addition, the inherent geographical distribution of development teams brings its own set of challenges.

The research has been carried out systematically and rigorously, which makes the observations and analyses presented in the thesis reliable. The main limitations of the general validity of the results are the relatively few number of cases that have been studied and the bias towards Swedish and western organizations and cultures; this bias can be discerned in the selection of cases studied as well as in the (partly unconscious) mindset of the author. It should also be remembered that at the beginning of this research the focus was on process aspects and software architecture; other approaches and viewpoints would likely give different types of answers.

The last research phase – aiming at validating and quantifying the results gathered so far – could easily be continued, as the data collection instrument and analysis guidelines have been designed and tested (in the form of the questionnaire and the analysis already performed and documented). With a larger number of cases, preferably involving more application domains, more national cultures – and more cases from the domains and cultures already represented – the results will be further confirmed. I also welcome studies of this topic from other viewpoints, such as those of organizational and decision-making psychology experts and those of other cultures.

At a more detailed level, there are a number of loose threads I consider would be challenging and interesting to pursue further. First, the observations made so far concerning the elicitation and documentation of the requirements of a future system are in my opinion very interesting, and worth further study as a separate topic. Second, the architectural patterns and styles of systems could be an additional indicator that the systems are

sufficiently similar for the *Merge* strategy to be practically possible. Third, the merge method and tool need to be evaluated for usefulness in realistic cases. Also, during their further development viewpoints or languages other than the simple module viewpoint currently implemented should be considered; in particular, by keeping a use-case view synchronized with other, more technical views, the architects could more easily communicate the impact of various alternative designs to the users.

The research topic is pertinent, and in the absence of relevant publications, we can conclude that it has not previously been studied systematically. The results presented in the thesis, which have been obtained using sound research methods, are of considerable potential value to the software engineering community. As the title promises, the thesis provides significant guidance in the use of a systematic engineering process for in-house software systems integration.

References

[1] Abowd G. D., Allen R., and Garlan D., "Using Style to Understand Descriptions of Software Architecture", In *Proceedings of The First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.

[2] Abrahamsson P., *The Role of Commitment in Software Process Improvement*, Ph.D. Thesis, Department of Information Processing Science, University of Oulu, 2005.

[3] Aggarwal K. K., Singh Y., and Chhabra J. K., "An Integrated Measure of Software Maintainability", In *Proceedings of Annual Reliability and Maintainability Symposium*, pp. 235-241, IEEE, 2002.

[4] Aho A., Sethi R., and Ullman J., *Compilers – Principles, Techniques and Tools*, Addison Wesely, 1986.

[5] Aiken P. H., *Data Reverse Engineering : Slaying the Legacy Dragon*, ISBN 0-07-000748-9, McGraw Hill, 1996.

[6] Allen R. and Garlan D., "A Formal Basis for Architectural Connection", In *ACM Transactions on Software Engineering and Methology*, volume 6, issue 3, pp. 213-249, 1997.

[7] Allen R., *A Formal Approach to Software Architecture*, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, 1997.

[8] Amazon, *Amazon.co.uk*, URL: http://www.amazon.co.uk/, 2006.

[9] Amazon, *Amazon.com*, URL: http://www.amazon.com/, 2006.

[10] Anderson K. M., Och C., King R., and Osborne R. M., "Integrating Infrastructure: Enabling Large-Scale Client Integration" , In *Proceedings of eleventh ACM Conference on Hypertext and Hypermedia*, pp. 57-66, ACM Press, 2000.

[11] Anderson K. M. and Sherba S. A., "Using XML to support Information Integration", In *Proceedings of International Workshop*

on XML Technologies and Software Engineering (XSE), IEEE, 2001.

[12] ANSI, ANSI, *American National Standards Institute*, http://www.ansi.org, 2004.

[13] April A., Huffman Hayes J., Abran A., and Dumke R., "Software Maintenance Maturity Model (SMmm): the software maintenance process model", In *Journal of Software Maintenance and Evolution: Research and Practice*, volume 17, issue 3, pp. 197-223, 2005.

[14] Ash D., Alderete J., Yao L., Oman P. W., and Lowther B., "Using software maintainability models to track code health", In *Proceedings of International Conference on Software Maintenance*, pp. 154-160, IEEE, 1994.

[15] Asklund U., "Identifying conflicts during structural merge", In *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research*, 1994.

[16] Association for Computing Machinery, *ACM Digital Library*, URL: http://portal.acm.org/, 2006.

[17] Astudillo H., Pereira J., and Lopéz C., "Evaluating Alternative COTS Assemblies from Unreliable Information", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2006.

[18] Atkinson C., Bunse C., Gross H.-G., and Peper C., *Component-Based Software Development for Embedded Systems : An Overview of Current Research Trends*, ISBN 3-540-30644-7, Springer, 2006.

[19] Bachman F., Bass L., Buhman S., Comella-Dorda S., Long F., Seacord R. C., and Wallnau K. C., *Volume II: Technical Concepts of Component-Based Software Engineering*, CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[20] Banzhaf W., Nordin P., Keller R. E., and Francone F. D., *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications*, ISBN 155860510X, Morgan Kaufmann, 1997.

[21] Bardram J., Christensen H. B., Corry A. V., Hansen K. M., and Ingstrup M., "Exploring Quality Attributes using Architectural Prototyping", In *Proceedings of First International Conference on the Quality of Software Architectures (QoSA 2005)*, Springer Verlag, 2005.

[22] Bardram J., Christensen H. B., and Hansen K. M., "Architectural Prototyping: An Approach for Grounding Architectural Design", In *Proceedings of 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, 2004.

[23] Basili V. R., Selby R. W., and Hutchens D. H., "Experimentation in Software Engineering", In *IEEE Transactions on Software Engineering*, volume 12, issue 7, pp. 733-734, 1986.

[24] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*, ISBN 0-201-19930-0, Addison-Wesley, 1998.

[25] Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.

[26] Beck K., *EXtreme Programming EXplained: Embrace Change*, ISBN 0201616416, Addison Wesley, 1999.

[27] Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R. C., Mellor S., Schwaber K., Sutherland J., and Thomas D., *Manifesto for Agile Software Development*, URL: http://agilemanifesto.org/, 2006.

[28] Beck K. and Fowler M., *Planning Extreme Programming*, ISBN 0201710919, Addison Wesley, 2000.

[29] Becker H. S., *Doing Things together: Selected Papers*, ISBN 0810107236, Northwestern University Press, 1986.

[30] Benestad H. C., Anda B., and Arisholm E., "Assessing Software Product Maintainability Based on Class-Level Structural Measures", In *Proceedings of 7th International Conference on Product-Focused Software Process Improvement (PROFES)*, pp. 94-111, Springer, 2006.

[31] Bengtsson P., *Architecture-Level Modifiability Analysis*, Ph.D. Thesis, Blekinge Institute of Technology, Sweden, 2002.

[32] Bernardo M., Ciancarini P., and Donatiello L., "Detecting architectural mismatches in process algebraic descriptions of software systems", In *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 77-86, IEEE, 2001.

[33] Berzins V., "On merging software extensions", In *Acta Informatica*, volume 23, pp. 607-619, 1986.

[34] Berzins V., “Software merge: semantics of combining changes to programs”, In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16, issue 6, pp. 1875-1903, 1994.

[35] Bhuta J. and Boehm B., “A Method for Compatible COTS Component Selection”, In *Proceedings of International Conference on COTS-Based Software Systems*, pp. 132-143, 2005.

[36] Blankers L., *Techniques and Processes for Assessing Compatibility of Third-Party Software Components*, M.Sc. Thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology (TU/e) and Department of Computer Science and Engineering, Mälardalen University, 2006.

[37] Boehm B., *Spiral Development: Experience, Principles and Refinements*, CMU/SEI-2000-SR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[38] Boehm B. and Turner R., *Balancing Agility and Discipline: A Guide for the Perplexed*, ISBN 0321186125, Addison-Wesley Professional, 2003.

[39] Booch G., *Object-Oriented Analysis and Design with Applications* (2nd edition), ISBN 0805353402, Benjamin/Cummings Publishing Company, 1994.

[40] Booch G., *Handbook of Software Architecture*, URL: http://www.booch.com/architecture/index.jsp, 2006.

[41] Booch G., Rumbaugh J., and Jacobson I., *The Unified Modeling Language User Guide*, ISBN 0201571684, Addison-Wesley, 1999.

[42] Bosch J., *Design & Use of Software Architectures*, ISBN 0-201-67494-7, Addison-Wesley, 2000.

[43] Bosch J., Gentleman M., Hofmeister C., and Kuusela J., “Preface”, in Bosch J., Gentleman M., Hofmeister C., and Kuusela J. (editors): *Software Architecture - System Design, Development and Maintenance, Third Working IEEE/IFIP Conference on Software Architecture (WICSA3)*, ISBN 1-4020-7176-0, Kluwer Academic Publishers, 2002.

[44] Bosch J., Molin P., Mattson M., and Bengtsson P., “Object Oriented Frameworks - Problems & Experiences”, in Fayad M.E., Schmidt D.C., and Johnson R.E. (editors): *Object-Oriented Application Frameworks*, Wiley & Sons, 1999.

[45] Bosch J. and Stafford J., “Architecting Component-based Systems”, in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[46] Bowman I. T., Holt R. C., and Brewster N. V., “Linux as a Case Study: Its Extracted Software Architecture”, In *Proceedings of 21st International Conference on Software Engineering (ICSE)*, 1999.

[47] Box D., *Essential COM*, ISBN 0-201-63446-5, Addison-Wesley, 1998.

[48] Brady J., Monk E., and Wagner B., *Concepts in Enterprise Resource Planning*, ISBN 0619015934, Course Technology, 2001.

[49] Bratthall L. G., van der Geest R., Hofmann H., Jellum E., Korendo Z., Martinez R., Orkisz M., Zeidler C., and Andersson J. S., “Integrating Hundred's of Products through One Architecture: the Industrial IT architecture”, In *Proceedings of the 24th International Conference on Software Engineering*, pp. 604-614, ACM, 2002.

[50] Bratthall L., Johansson E., and Regnell B., “Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution”, In *Proceedings of Second International Conference on Product Focused Software Process Improvement (PROFES)*, 2000.

[51] Britton C. and Bye P., *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems* (2nd edition), ISBN 0321246942, Pearson Education, 2004.

[52] Brodie M. L. and Stonebraker M., *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*, Morgan Kaufmann Series in Data Management Systems, ISBN 1558603301, Morgan Kaufmann, 1995.

[53] Brooks F. P., “No Silver Bullet”, in *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition*, ISBN 0201835959, Addison-Wesley Longman, 1995.

[54] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition* (20th Anniversary edition), ISBN 0201835959, Addison-Wesley Longman, 1995.

[55] Bub T. and Schwinn J., “VERBMOBIL: The Evolution of a Complex Large Speech-to-Speech Translation System”, In

Proceedings of Fourth International Conference on Spoken Language (ICSLP), pp. 2371-2374, IEEE, 1996.

[56] Buehler K. and Farley J. A., "Interoperability of Geographic Data and Processes: the OGIS Approach", In *StandardView*, volume 2, issue 3, pp. 163-168, 1994.

[57] Burrough P. A. and McDonnell R., *Principles of Geographical Information Systems* (2nd edition), ISBN 0198233655, Oxford University Press, 1998.

[58] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, John Wiley & Sons, 1996.

[59] Cannegieter J. J., "Controlling the Chaos of the CMMI Continuous Representation", In *Proceedings of 7th International Conference on Product-Focused Software Process Improvement (PROFES)*, Springer, 2006.

[60] Carmel E., *Global Software Teams - Collaborating Across Borders and Time Zones*, ISBN 0-13-924218-X, Prentice-Hall, 1999.

[61] Carmel E. and Agarwal R., "Tactical Approaches for Alleviating Distance in Global Software Development", In *IEEE Software*, volume 18, issue 2, pp. 22-29, 2001.

[62] Carmichael I., Tzerpos V., and Holt R. C., "Design maintenance: unexpected architectural interactions (experience report)", In *Proceedings of International Conference on Software Maintenance*, pp. 134-137, IEEE, 1995.

[63] Chalmers A. F., *What Is This Thing Called Science? An Assessment of the Nature and Status of Science and its Methods* (3rd edition), ISBN 0335201091, Hackett Publishing Company, 1999.

[64] Chapin N., "Software maintenance: a different view", In *Proceedings of AFIPS National Computer Conference*, pp. 509-513, AFIPS Press, 1985.

[65] Chester T. M., "Cross-Platform Integration with XML and SOAP", In *IT Professional*, volume 3, issue 5, pp. 26-34, 2001.

[66] Christensen M., Damm C. H., Hansen K. M., Sandvad E., and Thomsen M., "Design and evolution of software architecture in practice", In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS)*, pp. 2-15, 1999.

[67] Citeseer, *CiteSeer.IST Scientific Literature Digital Library*, URL: http://citeseer.ist.psu.edu/, 2006.

[68] Clements P., Kazman R., and Klein M., *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley, 2000.

[69] Clements P. and Northrop L., *Software Product Lines: Practices and Patterns*, ISBN 0-201-70332-7, Addison-Wesley, 2001.

[70] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Evaluating Software Architectures*, ISBN 0-201-70482-X, Addison-Wesley, 2001.

[71] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[72] Clément G., Larouche C., Gouin D., Morin P., and Kucera H., "OGDI: Toward Interoperability among Geospatial Databases", In *ACM SIGMOD Record*, volume 26, issue 3, pp. 108-, 1997.

[73] CMMI Product Team, *Capability Maturity Model ® Integration (CMMI SM), Version 1.1*, CMU/SEI-2002-TR-011, Software Engineering Institute (SEI), 2002.

[74] Coleman D., Booch G., Garlan D., Iyengar S., Kobryn C., and Stavridou V., *Is UML an Architectural Description Language?*, Panel at Conference on Object-Oriented Programming, Systems, Languages, and applications (OOPSLA) 1999, URL: http://www.acm.org/sigs/sigplan/oopsla/oopsla99/2_ap/tech/2d1a_uml.html, 2003.

[75] Coleman D., Ash D., Lowther B., and Oman P., "Using Metrics to Evaluate Software System Maintainability", In *IEEE Computer*, volume 27, issue 8, pp. 44-49, 1994.

[76] Collins H. M. and Pinch T., *The Golem : What You Should Know about Science* (2nd edition), ISBN 0521645506, Cambridge University Press, 1998.

[77] Conway M. E., "How do committees invent?", In *Datamation*, volume 14, issue 4, pp. 28-31, 1968.

[78] Cormen T. H., Leiserson C. E., and Rivest R. L., *Introduction to Algorithms*, ISBN 0-262-53091-0, MIT Press, 1990.

[79] Correia J. M., Biscotti F., and Dharmasthira Y., *Forecast: AIM and Portal Software, Worldwide, 2005-2010*, Gartner, 2006.

[80] Crnkovic Ivica and Larsson M., "Challenges of Component-based Development", In *Journal of Systems & Software*, volume 61, issue 3, pp. 201-212, 2002.

[81] Crnkovic I., Asklund U., and Persson-Dahlqvist A., *Implementing and Integrating Product Data Management and Software Configuration Management*, ISBN 1-58053-498-8, Artech House, 2003.

[82] Crnkovic I. and Chaudron M., "Component-based Development Process and Component Lifecycle", In *Proceedings of 27th International Conference Information Technology Interfaces (ITI)*, IEEE, 2005.

[83] Crnkovic I. and Larsson M., *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[84] Cummins F. A., *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*, ISBN 0471400106, John Wiley & Sons, 2002.

[85] Curtis B., Hefley B., and Miller S., *People Capability Maturity Model (P-CMM)*, CMU/SEI-2001-MM-001, Software Engineering Insitute, 2005.

[86] Dagdeviren H., Juric R., and Kassana T. A., "An Exploratory Study for Effective COTS and OSS Product Marketing", In *Proceedings of 27th International Conference on Information Technology Interfaces (ITI)*, pp. 681-686, IEEE, 2005.

[87] Dashofy E. M. and van der Hoek A., "Representing Product Family Architectures in an Extensible Architecture Description Language", In *Proceedings of The International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain*, 2001.

[88] Davis H. C., Millard D. E., Reich S., Bouvin N., Grønbæk K., Nürnberg P. J., Sloth L., Wiil U. K., and Anderson K., "Interoperability between Hypermedia Systems: The Standardisation Work of the OHSWG", In *Proceedings of Tenth ACM Conference on Hypertext and Hypermedia: Returning to Our Diverse Roots*, pp. 201-202, ACM, 1999.

[89] Davis L., Flagg D., Gamble R. F., and Karatas C., "Classifying Interoperability Conflicts", In *Proceedings of Second International Conference on COTS-Based Software Systems* , pp. 62-71, LNCS 2580, Springer Verlag, 2003.

[90] Davis L., Gamble R., Payton J., Jónsdóttir G., and Underwood D., "A Notation for Problematic Architecture Interactions", In *ACM SIGSOFT Software Engineering Notes*, volume 26, issue 5, 2001.

[91] Davis L. A. and Gamble R. F., "Identifying Evolvability for Integration", In *Proceedings of First International Conference on COTS-Based Software Systems (ICCBSS)*, pp. 65-75, Springer, 2002.

[92] Decker S., Melnik S., van Harmelen F., Fensel D., Klein M., Broekstra J., Erdmann M., and Horrocks I., "The Semantic Web: The Roles of XML and RDF", In *IEEE Internet Computing*, volume 4, issue 5, pp. 63-74, 2000.

[93] DeLine R., "Avoiding Packaging Mismatch with Flexible Packaging", In *IEEE Transactions on Software Engineering*, volume 27, issue 2, pp. 124-143, 20010.

[94] DeMarco T. and Lister T., *Peopleware : Productive Projects and Teams* (2nd edition), ISBN 0-932633-43-9, Dorset House Publishing, 1999.

[95] Denning P. J. and Dargan P. A., "A discipline of software architecture", In *ACM Interactions*, volume 1, issue 1, 1994.

[96] Denzin N. K., *The Research Act: A Theoretical Introduction to Sociological Methods* (3rd edition), Prentice-Hall, 1989.

[97] DePrince W. and Hofmeister C., "Analyzing Commercial Component Models", In *Proceedings of Third Working IEEE/IFIP Conference on Software Architecture (WICSA3)*, pp. 205-219, Kluwer Academic Publishers, 2002.

[98] Di Marco A. and Mirandola R., "Model Transformation in Software Performance Engineering", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2006.

[99] Dijkstra E. W., "The Structure of the THE Multiprogramming System", In *Communications of the ACM*, volume 11, issue 5, pp. 341-346, 1968.

[100] Dikel D. M., Kane D., and Wilson J. R., *Software Architecture - Organizational Principles and Patterns*, Software Architecture Series, ISBN 0-13-029032-7, Prentice Hall PTR, 2001.

[101] Duzbayev N. and Poernomo I., "Runtime prediction of queued behaviour", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2006.

[102] Egyed A., Medividovic N., and Gacek C., "Component-based perspective on software mismatch detection and resolution", In *IEE Proceedings - Software*, volume 147, issue 6, 2000.

[103] EIA, *Systems Engineering Capability Model*, EIA Standard 731.1, Electronic Industries Alliance, 2002.

[104] Emmerich W., Ellmer E., and Fieglein H., "TIGRA - An Architectural Style for Enterprise Application Integration", In *Proceedings of 23rd International Conference on Software Engineering*, pp. 567-576, IEEE, 2001.

[105] Estublier J. and Favre J.-M., "Component Models and Technology", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[106] Fayad M. E., Hamu D. S., and Brugali D., "Enterprise frameworks characteristics, criteria, and challenges", In *Communications of the ACM*, volume 43, issue 10, pp. 39-46, 2000.

[107] Fayad M. E. and Schmidt D. C., "Object-oriented application frameworks", In *Communications of the ACM*, volume 40, issue 10, pp. 32-38, 1997.

[108] Ferneley E. H., "Design Metrics as an Aid to Software Maintenance: An Empirical Study", In *Journal of Software Maintenance: Research and Practice*, volume 11, issue 1, pp. 55-72, 1999.

[109] Fowler M., *Refactoring: Improving the Design of Existing Code*, ISBN 0201485672, Addison-Wesley, 1998.

[110] Fowler M., *Patterns of Enterprise Application Architecture*, ISBN 0321127420, Addison-Wesley, 2002.

[111] Fowler M., "Who Needs an Architect?", In *IEEE Software*, volume 20, issue 5, pp. 11-13, 2003.

[112] Gacek C., *Detecting Architectural Mismatches During Systems Composition*, USC/CSE-97-TR-506, University of Southern California, 1997.

[113] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2, Addison-Wesley, 1995.

[114] Garlan D., Allen R., and Ockerbloom J., “Exploiting Style in Architectural Design Environments”, In *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng*, 1994.

[115] Garlan D., Allen R., and Ockerbloom J., “Architectural Mismatch: Why Reuse is so Hard”, In *IEEE Software*, volume 12, issue 6, pp. 17-26, 1995.

[116] Garlan D., Monroe R. T., and Wile D., “Acme: Architectural Description of Component-Based Systems”, in Leavens G.T. and Sitarman M. (editors): *Foundations of Component-Based Systems*, Cambridge University Press, 2000.

[117] Garlan D. and Shaw M., “An Introduction to Software Architecture”, In *Advances in Software Engineering and Knowledge Engineering*, volume I, 1993.

[118] GEIA, *Processes for Engineering a System*, EIA-632, Government Electronics and Information Technology Association, 1999.

[119] Ghezzi C., Jazayeri M., and Mandrioli D., *Fundamentals of Software Engineering*, ISBN 0-13-305699-6, Prentice Hall, Pearson Education, 2003.

[120] Gilb T., *Principles of Software Engineering Management*, ISBN 0-201-19246-2, Addison-Wesley, 1988.

[121] Google, *Google*, URL: http://www.google.com/, 2006.

[122] Gorton I., Thurman D., and Thomson J., “Next Generation Application Integration Challenges and New Approaches”, In *Proceedings of 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pp. 585-590, IEEE, 2003.

[123] Grady J. O., *Systems Integration*, CRC Press, 1994.

[124] Grady R. B., “Successfully Applying Software Metrics”, In *IEEE Computer*, volume 27, issue 9, pp. 18-25, 1994.

[125] Gremba J. and Myers C., *The IDEAL Model: A Practical Guide for Improvement*, Bridge, issue three, Software Engineering Institute (SEI), 1997.

[126] Griss M. L., Favaro J., and Walton P., “Managerial and organizational issues - starting and running a software reuse program”, in Schäfer W., Prieto-díaz R., and Matsumoto M. (editors): *Software Reusability*, ISBN 0-13-063918-4, Ellis Horwood, 1994.

[127] Gröne B., Knöpfel A., and Kugel R., “Architecture recovery of Apache 1.3 - A case study”, In *Proceedings of International Conference on Software Engineering Research and Practice*, CSREA Press, 2002.

[128] Guarino N., *Formal Ontology in Information Systems*, ISBN 9051993994, IOS Press, 1998.

[129] Gyllenswärd E., Kap M., and Land R., “Information Organizer - A Comprehensive View on Reuse”, In *Proceedings of 4th International Conference on Enterprise Information Systems (ICEIS)*, 2002.

[130] Halsall F., *Data Communications, Computer Networks, and Open Systems* (4th edition), ISBN 020142293X, Addison-Wesley, 1996.

[131] Halstead M. H., *Elements of Software Science*, Operating, and Programming Systems Series, Elsevier, 1977.

[132] Hamlet D. and Maybee J., *The Engineering of Software*, ISBN 0-201-70103-0, Addison Wesley Longman, 2001.

[133] Harold E. R. and Means W. S., *XML in a Nutshell* (2nd edition), ISBN 0596002920, O'Reilly, 2004.

[134] Hasselbring W., “Information System Integration”, In *Communications of the ACM*, volume 43, issue 6, pp. 33-38, 2000.

[135] Hasso Plattner Institute (HPI), *Apache Modeling Portal*, URL: http://apache.hpi.uni-potsdam.de/, 2003.

[136] Hasso Plattner Institute (HPI), *Fundamental Modeling Concepts (FMC) Web Site*, URL: http://fmc.hpi.uni-potsdam.de/, 2003.

[137] Heiler S., “Semantic Interoperability”, In *ACM Computing Surveys*, volume 27, issue 2, pp. 271-273, 1995.

[138] Heineman G. T. and Councill W. T., *Component-based Software Engineering, Putting the Pieces Together*, ISBN 0-201-70485-4, Addison-Wesley, 2001.

[139] Heinisch C. and Goll J., “Consistent Object-Oriented Modeling of System Dynamics with State-Based Collaboration Diagrams”, In *Proceedings of International Conference on Software Engineering Research and Practice*, CSREA Press, 2003.

[140] Henry S. and Kafura D., “Software Structure Metrics Based on Information Flow”, In *IEEE Transactions on Software Engineering*, volume SE7, issue 5, pp. 510-519, 1981.

[141] Herbsleb J. D. and Moitra D., "Global Software Development", In *IEEE Software*, volume 18, issue 2, pp. 16-20, 2001.

[142] Hermansson H., Johansson M., and Lundberg L., "A Distributed Component Architecture for a Large Telecommunication Application", In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, pp. 188-195, 2000.

[143] Hissam S. A., Moreno G. A., Stafford J., and Wallnau K. C., *Packaging Predictable Assembly with Prediction-Enabled Component Technology*, Technical report CMU/SEI-2001-TR-024 ESC-TR-2001-024, 2001.

[144] Hissam S. A., Hudak J., Ivers J., Klein M., Larsson M., Moreno G. A., Northrop L., Plakosh D., Stafford J., Wallnau K. C., and Wood W., *Predictable Assembly of Substation Automation Systems: An Experience Report*, CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2002.

[145] Hissam S. A., Hudak J., Ivers J., Klein M., Larsson M., Moreno G. A., Northrop L., Plakosh D., Stafford J., Wallnau K. C., and Wood W., *Predictable Assembly of Substation Automation Systems: An Experience Report, Second Edition*, CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2003.

[146] Hissam S. A., Moreno G. A., Stafford J., and Wallnau K. C., "Enabling Predictable Assembly", In *Journal of Systems & Software*, volume 65, issue 3, pp. 185-198, 2003.

[147] Hissam S. A., Stafford J., and Wallnau K. C., *Volume III: Anatomy of a Reasoning-Enabled Component Technology*, CMU/SEI-2001-TR-007, Software Engineering Institute, Carnegie Mellon University, 2001.

[148] Hofmeister C. and Nord R., "From software architecture to implementation with UML", pp. 113-114, IEEE, 2001.

[149] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[150] Hofstede G., *Cultures and Organizations: Software of the Mind* (2nd edition), ISBN 0071439595, McGraw-Hill, 2004.

[151] Hohmann L., *Beyond Software Architecture*, The Addison-Wesley Signature Series, ISBN 0-201-77594-8, Addison-Wesley, 2003.

[152] Hunter J., "Enhancing the Semantic Interoperability of Multimedia Through a Core Ontology", In *IEEE Transactions on Circuits & Systems for Video Technology*, volume 13, issue 1, pp. 49-59, 2003.

[153] IASA, *International Association of Software Architects*, URL: http://www.iasahome.org, 2006.

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

[155] IEEE, Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, volume 21, issue 4, 1995.

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

[157] IEEE, IEEE, *IEEE Standards Association Home Page*, http://standards.ieee.org/, 2004.

[158] IEEE, *IEEE Xplore*, URL: http://ieeexplore.ieee.org/, 2006.

[159] IEEE Architecture Working Group, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE, 2000.

[160] ISO, ISO, *ISO - International Organization for Standardization*, http://www.iso.org, 2004.

[161] ISO/IEC, *Information Technology - Software Life-Cycle Processes*, ISO/IEC 12207:1995 (E), ISO/IEC, 1995.

[162] ISO/IEC, *Systems Engineering - System life cycle processes*, ISO/IEC 15288:2002(E), ISO/IEC, 2002.

[163] Jacobson I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ISBN 0201544350, Addison-Wesley, 1992.

[164] Jakobsson L., Christiansson B., and Crnkovic I., "Component-Based Development Process", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[165] Jaktman C. B., Leaney J., and Liu M., "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", In *Proceedings of The First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer Academic Publishers, 1999.

[166] Jansen A. and Bosch J., "Software Architecture as a Set of Architectural Design Decisions", In *Proceedings of 5th Working*

IEEE/IFIP Conference on Software Architecture, pp. 109-118, IEEE, 2005.

[167] Johansson E. and Höst M., "Tracking Degradation in Software Product Lines through Measurement of Design Rule Violations", In *Proceedings of 14th International Conference in Software Engineering and Knowledge Engineering (SEKE)*, ACM, 2002.

[168] John I., Muthig D., Sody P., and Tolzmann E., "Efficient and systematic software evolution through domain analysis", In *Requirements Engineering, 2002.Proceedings.IEEE Joint International Conference on*, pp. 237-244, 2002.

[169] Johnson P., *Enterprise Software System Integration - An Architectural Perspective*, Ph.D. Thesis, Industrial Information and Control Systems, Royal Institute of Technology, 2002.

[170] Johnson R. E., "Frameworks = (Components + Patterns)", In *Communications of the ACM*, volume 40, issue 10, pp. 39-42, 1997.

[171] Kajko-Mattson M., "Preventive Maintenance! Do we know what it is?", In *Proceedings of International Conference on Software Maintenance (ICSM)*, IEEE, 2000.

[172] Kajko-Mattson M., "Problems within Support (Upfront Maintenance)", In *Proceedings of Seventh European Conference on Software Maintenance and Reengineering*, IEEE, 2003.

[173] Kajko-Mattsson M., *Corrective Maintenance Maturity Model: Problem Management*, Department of Computer and Systems Sciences, Stockholm University and Royal Institute of Technology, 2001.

[174] Kajko-Mattsson M., "Motivating the Corrective Maintenance Maturity Model", In *Proceedings of Seventh IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, 2001.

[175] Kaplan A., Ridgway J., and Wileden J. C., "Why IDLs are Not Ideal", In *Proceedings of 9th International Workshop on Software Specification and Design*, ACM, 1998.

[176] Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.

[177] Karlsson L. and Regnell B., "Introducing Tool Support for Retrospective Analysis of Release Planning Decisions", In

Proceedings of 7th International Conference on Product Focused Software Process Improvement (PROFES), pp. 19-33, Springer, 2006.

[178] Karolak D. W., *Global Software Development - Managing Virtual Teams and Environments*, ISBN 0-8186-8701-0, IEEE Computer Society, 1998.

[179] Kazman R., "Tool support for architecture analysis and design", In *Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops (jointly)*, pp. 94-97, 1996.

[180] Kazman R., Abowd G., Bass L., and Clements P., "Scenario-Based Analysis of Software Architecture", In *IEEE Software*, volume 13, issue 6, pp. 47-55, 1996.

[181] Kazman R., Barbacci M., Klein M., and Carriere J., "Experience with Performing Architecture Tradeoff Analysis Method", In *Proceedings of The International Conference on Software Engineering, New York*, pp. 54-63, 1999.

[182] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures", In *Proceedings of The 16th International Conference on Software Engineering*, 1994.

[183] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., and Carriere J., "The Architecture Tradeoff Analysis Method", In *Proceedings of The Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 68-78, IEEE, 1998.

[184] Keller F., Tabeling P., Apfelbacher R., Gröne B., Knöpfel A., Kugel R., and Schmidt O., "Improving Knowledge Transfer at the Architectural Level: Concepts and Notations", In *Proceedings of International Conference on Software Engineering Research and Practice*, CSREA Press, 2002.

[185] Keller F. and Wendt S., "FMC: An Approach Towards Architecture-Centric System Development", In *Proceedings of 10th IEEE Symposium and Workshops on Engineering of Computer Based Systems*, IEEE, 2003.

[186] Keshav R. and Gamble R., “Towards a Taxonomy of Architecture Integration Strategies”, In *Proceedings of third International Workshop on Software Architecture*, pp. 89-92, ACM, 1998.

[187] Kim D.-H. and Kim M.-S., “Web GIS Service Component Based On Open Environment”, In *Proceedings of IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pp. 3346-3348, IEEE, 2002.

[188] Kindrick J. D., Sauter J. A., and Matthews R. S., “Improving conformance and interoperability testing”, In *StandardView*, volume 4, issue 1, 1996.

[189] Kircher M. and Prashant J., *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management* , ISBN 0-470-84525-2, Wiley, 2004.

[190] Kitchenham B., *Procedures for Performing Systematic Reviews*, TR/SE0401, Keele University, 2004.

[191] Kitchenham B. A., Travassos G. H., von Mayrhauser A., Niessink F., Schneidewind N. F., Singer J., Takada S., Vehvilainen R., and Yang H., “Towards and Ontology of Software Maintenance”, In *Journal of Software Maintenance: Research and Practice*, volume 11, issue 6, pp. 365-389, 1999.

[192] Kitchenham B., Pickard L., and Pfleeger S. L., “Case Studies for Method and Tool Evaluation”, In *IEEE Software*, volume 12, issue 4, pp. 52-62, 1995.

[193] Klein J., “How Does the Architect's Role Change as the Software Ages?”, In *Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, 2005.

[194] Kobryn C., “Modeling enterprise software architectures using UML”, In *Proceedings of Second International Enterprise Distributed Object Computing Workshop*, pp. 25-34, 1998.

[195] Korhonen M. and Mikkonen T., “Assessing Systems Adaptability to a Product Family”, In *Proceedings of International Conference on Software Engineering Research and Practice*, CSREA Press, 2003.

[196] Kotonya G. and Sommerville I., *Requirements Engineering: Processes and Techniques*, ISBN 0471972088, John Wiley & Sons, 1998.

[197] Kramer R. and Sesink L., “Framework for Photographic Archives Interoperability”, In *Proceedings of The 3rd Conference on*

Standardization and Innovation in Information Technology, pp. 135-140, IEEE, 2003.

[198] Kruchten P., Selic B., and Kozaczynski W., "Tutorial: describing software architecture with UML", pp. 693-694, ACM, 2002.

[199] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, pp. 42-50, 1995.

[200] Kruchten P., *The Rational Unified Process: An Introduction* (2nd edition), ISBN 0-201-70710-1, Addison-Wesley, 2000.

[201] Kruchten P., Lago P., and van Vliet H., "Building up and Reasoning about Architectural Knowledge", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2006.

[202] Kuhn T. S., *The Structure of Scientific Revolutions* (3rd edition), ISBN 0-226-45808-3, The University of Chicago Press, 1996.

[203] Laitinen K., "Estimating Understandability of Software Documents", In *ACM SIGSOFT Software Engineering Notes* , volume 21, issue 4, pp. 81-92, 1996.

[204] Land R., *Architectural Solutions in PAM*, M.Sc. Thesis, Department of Computer Engineering, Mälardalen University, 2001.

[205] Land R., "Software Deterioration And Maintainability – A Model Proposal", In *Proceedings of Second Conference on Software Engineering Research and Practise in Sweden (SERPS)*, pp. X-Y, Blekinge Institute of Technology Research Report 2002:10, 2002.

[206] Land R., *An Architectural Approach to Software Evolution and Integration*, Licentiate Thesis, Department of Computer Science and Engineering, Mälardalen University, 2003.

[207] Land R., "Applying the IEEE 1471-2000 Recommended Practice to a Software Integration Project", In *Proceedings of International Conference on Software Engineering Research and Practice (SERP'03)*, CSREA Press, 2003.

[208] Land R., *Interviews on Software Systems Merge*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-196/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[209] Land R., Blankers L., Larsson S., and Crnkovic I., "Software Systems In-House Integration Strategies: Merge or Retire - Experiences from Industry", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, 2005.

[210] Land R., Carlson J., Crnkovic I., and Larsson S., "A Method for Exploring Software Systems Merge Alternatives", In *Proceedings of submitted to Quality of Software Architectures (QoSA) (will otherwise be published as a technical report, the paper can be found at www.idt.mdh.se/~rld/temp/MergeMethod.pdf)*, 2006.

[211] Land R. and Crnkovic I., "Software Systems Integration and Architectural Analysis - A Case Study", In *Proceedings of International Conference on Software Maintenance (ICSM)*, IEEE, 2003.

[212] Land R. and Crnkovic I., "Existing Approaches to Software Integration – and a Challenge for the Future", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, Linköping University, 2004.

[213] Land R. and Crnkovic I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", In *Information & Software Technology*, volume Accepted for publication, 2006.

[214] Land R., Crnkovic I., and Larsson S., "Concretizing the Vision of a Future Integrated System – Experiences from Industry", In *Proceedings of 27th International Conference Information Technology Interfaces (ITI)*, IEEE, 2005.

[215] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Concerns When Selecting an In-House Integration Strategy - Experiences from Industry", In *Proceedings of 5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, IEEE, 2005.

[216] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Reuse in Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of First International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2005.

[217] Land R., Crnkovic I., and Wallin C., "Integration of Software Systems - Process Challenges", In *Proceedings of Euromicro Conference*, 2003.

[218] Land R. and Lakotic M., "A Tool for Exploring Software Systems Merge Alternatives", In *Proceedings of International ERCIM Workshop on Software Evolution* , 2006.

[219] Land R., Larsson S., and Crnkovic I., *Interviews on Software Integration*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-177/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2005.

[220] Land R., Larsson S., and Crnkovic I., "Processes Patterns for Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, 2005.

[221] Land R., Thilenius P., Larsson S., and Crnkovic I., *A Quantitative Survey on Software In-house Integration*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-203/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[222] Land R., Thilenius P., Larsson S., and Crnkovic I., "Software In-House Integration – Quantified Experiences from Industry", In *Proceedings of 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, IEEE, 2006.

[223] Lange R. and Schwanke R. W., "Software Architecture Analysis: A Case Study", In *Proceedings of 3rd international workshop on Software configuration management*, ACM, 1991.

[224] Lanning D. L. and Khoshgoftaar T. M., "Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty", In *IEEE Computer*, volume 27, issue 9, pp. 35-40, 1994.

[225] Larsson M., *Applying Configuration Management Techniques to Component-Based Systems*, Licentiate Thesis, Dissertation 2000-007, Department of Information Technology Uppsala University., 2000.

[226] Larsson M., *Predicting Quality Attributes in Component-based Software Systems*, Ph.D. Thesis, Mälardalen University, 2004.

[227] Larsson M. and Crnkovic I., "New Challenges for Configuration Management", In *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1675, Springer Verlag, 1999.

[228] Larsson S., *Improving Software Product Integration*, Technology Licentiate Thesis, Department of Computer Science and Electronics, Mälardalen University, 2005.

[229] Laudon K. C. and Laudon J. P., *Management Information Systems* (8th edition), ISBN 0131014986, Pearson Education, 20030.

[230] Le H. and Howlett C., "Client-Server Communication Standards for Mathematical Computation", In *Proceedings of International Conference on Symbolic and Algebraic Computation*, pp. 299-306, ACM Press, 1999.

[231] Leclercq E., Benslimane D., and Yétongnon K., "ISIS: A Semantic Mediation Model and an Agent Based Architecture for GIS Interoperability", In *Proceedings of International Symposium Database Engineering and Applications (IDEAS)*, pp. 87-91, IEEE, 1999.

[232] Lee J., "Design rationale systems: understanding the issues", In *IEEE Expert*, volume 12, issue 3, pp. 78-85, 1997.

[233] Lee J., Siau K., and Hong S., "Enterprise Integration with ERP and EAI", In *Communications of the ACM*, volume 46, issue 2, pp. 54-60, 2003.

[234] Lehman M. M., Perry D. E., and Ramil J. F., "Implications of evolution metrics on software maintenance", In *Proceedings of International Conference on Software Maintenance*, pp. 208-217, IEEE, 1998.

[235] Lehman M. M. and Ramil J. F., *FEAST project*, URL: http://www.doc.ic.ac.uk/~mml/feast/, 2001.

[236] Lehman M. M. and Ramil J. F., "Rules and Tools for Software Evolution Planning and Management", In *Annals of Software Engineering*, volume 11, issue 1, pp. 15-44, 2001.

[237] Lehman M. M. and Ramil J. F., "Software Evolution and Software Evolution Processes", In *Annals of Software Engineering*, volume 14, issue 1-4, pp. 275-309, 2002.

[238] Li M., Puder A., and Schieferdecker I., "A Test Framework for CORBA Interoperability", In *Proceedings of Fifth IEEE International Enterprise Distributed Object Computing Conference*, pp. 152-161, IEEE, 2001.

[239] Libicki M., "Second-Best Practices for Interoperability", In *StandardView*, volume 4, issue 1, pp. 32-35, 1996.

[240] Lientz B. P. and Swanson E. B., *Software Maintenance Management*, Addison-Wesley, 1980.

[241] Linthicum D. S., *Enterprise Application Integration*, Addison-Wesley Information Technology Series, ISBN 0201615835, Addison-Wesley, 1999.

[242] Linthicum D. S., *B2B Application Integration: e-Business-Enable Your Enterprise*, ISBN 0201709368, Addison-Wesley, 2003.

[243] Longley P. A., Goodchild M. F., Maguire D. J., and Rhind D. W., *Geographic Information Systems and Science*, ISBN 0471892750, John Wiley & Sons, 2001.

[244] Losavio F., Ortega D., and Perez M., "Modeling EAI", In *Proceedings of 12th International Conference of the Chilean Computer Science Society*, pp. 195-203, IEEE, 2002.

[245] Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., and Mann W., "Specification and Analysis of System Architecture Using Rapide", In *IEEE Transactions on Software Engineering*, issue Special Issue on Software Architecture, pp. 336-335, 1995.

[246] Lung C.-H., "Software architecture recovery and restructuring through clustering techniques", In *Proceedings of Third International Workshop on Software Architecture (ISAW)*, pp. 101-104, ACM Press, 1998.

[247] Lung C.-H., Bot S., Kalaichelvan K., and Kazman R., "An Approach to Software Architecture Analysis for Evolution and Reusability", In *Proceedings of Centre for Advanced Studies Conference (CASCON)*, pp. 144-154, 1997.

[248] Lüders F., *Use of Component-Based Software Architectures in Industrial Control Systems*, Technology Licentiate Thesis, Mälardalen University, Sweden, 2003.

[249] Lüders F., Lau K.-K., and Ho S.-M., "On the Specification of Components", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[250] Maccari A. and Galal G. H., "Introducing the Software Architectonic Viewpoint", In *Proceedings of Third Working IEEE/IFIP Conference on Software Architecture (WICSA3)*, pp. 175-189, Kluwer Academic Publishers, 2002.

[251] Maciaszek L. A. and Liong B. L., *Practical Software Engineering*, ISBN 0 321 20465 4, Pearson Education Limited, 2005.

[252] Malveau R. and Mowbray T. J., *Software Architect Bootcamp*, Software Architecture Series, ISBN 0-13-027407-0, Prentice Hall PTR, 2001.

[253] Mattson M., Bosch J., and Fayad M. E., "Framework Integration Problems, Causes, Solutions", In *Communications of the ACM*, volume 42, issue 10, pp. 81-87, 1999.

[254] Maxwell Joseph A., "Understanding and validity in qualitative research", In *Harvard Educational Review*, volume 62, issue 3, pp. 279-300, 1992.

[255] Mazen M. and Dibuz S., "Pragmatic method for interoperability test suite derivation", In *Proceedings of 24th Euromicro Conference*, pp. 838-844, IEEE, 1998.

[256] McCabe T. J., "A Complexity Measure", In *IEEE Transaction on Software Engineering*, volume 2, pp. 308-320, 1976.

[257] McConnell S., *Rapid Development, Taming Wild Software Schedules*, ISBN 1-55615-900-5, Microsoft Press, 1996.

[258] Medvidovic N., "On the Role of Middleware in Architecture-Based Software Development", In *Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 299-306, ACM Press, 2002.

[259] Medvidovic N., Rosenblum D. S., Redmiles D. F., and Robbins J. E., "Modeling Software Architectures in the Unified Modeling Language", In *ACM Transactions on Software Engineering and Methodology*, volume 11, issue 1, pp. 2-57, 2002.

[260] Medvidovic N., Rosenblum D. S., and Taylor R. N., "An Architecture-Based Approach to Software Evolution", In *Proceedings of International Workshop on the Principles of Software Evolution (IWPSE)*, IEEE, 1999.

[261] Medvidovic N. and Taylor R. N., "A classification and comparison framework for software architecture description languages", In *IEEE Transactions on Software Engineering*, volume 26, issue 1, pp. 70-93, 2000.

[262] Mehta A. and Heineman G. T., "Evolving legacy system features into fine-grained components", In *Proceedings of 24th International Conference on Software Engineering*, pp. 417-427, 2002.

[263] Mens T., “A state-of-the-art survey on software merging”, In *IEEE Transactions on Software Engineering*, volume 28, issue 5, pp. 449-462, 2002.

[264] Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirschfeld R., and Jazayeri M., “Challenges in Software Evolution”, In *Proceedings of Eigth International Workshop on Principles of Software Evolution (IWPSE)*, IEEE, 2005.

[265] Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0-201-70454-4, Addison-Wesley, 2001.

[266] Michell M., *An Introduction to Genetic Algorithms (Complex Adaptive Systems)* (Reprint edition), ISBN 0262631857, MIT Press, 1998.

[267] Millard D. E., Moreau L., Davis H. C., and Reich S., “FOHM: a Fundamental Open Hypertext Model for Investigating Interoperability between Hypertext Domains”, In *Proceedings of Eleventh ACM Conference on Hypertext and Hypermedia*, pp. 93-102, ACM, 2000.

[268] Mitchell B., Traverso M., and Mancoridis S., “An architecture for distributing the computation of software clustering algorithms”, In *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 181-190, 2001.

[269] Monson-Haefel R., *Enterprise JavaBeans* (3rd edition), ISBN 0596002262, O'Reilly & Associates, 2001.

[270] Moreno G. A., Hissam S. A., and Wallnau K. C., “Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling”, In *Proceedings of 5th Workshop on component based software engineering*, 2002.

[271] Muskens J., Bril R. J., and Chaudron M. R. V., “Generalizing Consistency Checking between Software Views”, In *Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 169-178, IEEE, 2005.

[272] Mustapic G., *Architecting Software for Complex Embedded Systems - Quality Attribute Based Approach to Openness*, Department of Computer Science and Engineering, Mälardalen University, 2004.

[273] Mustapic G., Wall A., Norström C., Crnkovic I., Sandström K., Fröberg J., and Andersson J., “Real World Influences on Software

Architecture - Interviews with Industrial Software Experts", In *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 101-111, IEEE, 2004.

[274] Myers M. D., *Qualitative Research in Information Systems*, URL: http://www.qual.auckland.ac.nz/, 2006.

[275] Ncube C. and Maiden N. A., "Selecting the Right COTS Software: Why Requirements Are Important", in Heineman G.T. and Councill W.T. (editors): *Component-Based Software Engineering: Putting the Pieces Together*, ISBN 0-201-70485-4, Addison-Wesley, 2001.

[276] Neale J. M. and Liebert R. M., *Science and Behaviour - An Introduction to Methods of Research* (3rd edition), ISBN 0137951396, Prentice Hall International, Inc, 1985.

[277] Nedstam J., *Strategies for Management of Architectural Change and Evolution*, Ph.D. Thesis, Department of Communication Systems, Faculty of Engineering, Lund University, 2005.

[278] Newton H., *Newton's Telecom Dictionary: Covering Telecommunications, Networking, Information Technology, Computing and the Internet* (20th edition), ISBN 1578203090, CMP Books, 2004.

[279] Niazi M., Wilson D., and Zowghi D., "Implementing Software Process Improvement Initiatives: An Empirical Study", In *Proceedings of 7th International Conference on Product-Focused Software Process Improvement (PROFES)*, pp. 222-233, Springer, 2006.

[280] Nielsen J., *Usability Engineering*, ISBN 0-125-18406-9, Academic Press, 1993.

[281] Nordby E. and Blom M., "Semantic Integrity in CBD", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[282] O'Leary D. E., *Enterprise Resource Planning Systems: Systems, Life Cycle, Electronic Commerce, and Risk* (1st edition), ISBN 0521791529, Cambridge University Press, 2000.

[283] OGC, OGC, *Open GIS Consortium, Inc.*, http://www.opengis.org/, 2004.

[284] Olsson K. and Karlsson E.-A., *Daily Build - The Best of Both Worlds Rapid Development and Control*, V040083, Sveriges Verkstadsindustrier, 1999.

[285] Oman P. and Hagemeister J., “Metrics for Assessing a Software System's Maintainability”, In *Proceedings of Conference on Software Maintenance*, pp. 337-344, IEEE, 1992.

[286] Oman P., Hagemeister J., and Ash D., *A Definition and Taxonomy for Software Maintainability*, SETL Report 91-08-TR, University of Idaho, 1991.

[287] Omelayenko B., “Integration of Product Ontologies for B2B Marketplaces: A Preview”, In *ACM SIGecom Exchanges* , volume 2, issue 1, pp. 19-25, 2000.

[288] OMG, *Object Management Group* , URL: http://www.omg.org, 2003.

[289] OMG, *The Open Group Architectural Framework*, URL: http://www.opengroup.org/architecture/togaf8-doc/arch/, 2003.

[290] OMG, *UML 2.0 Standard Officially Adopted at OMG Technical Meeting in Paris*, URL: http://www.omg.org/news/releases/pr2003/6-12-032.htm, 2003.

[291] Open Group T., *ADML Preface*, URL: http://www.opengroup.org/onlinepubs/009009899/, 2003.

[292] Oreizy P., “Decentralized Software Evolution”, In *Proceedings of International Conference on the Principles of Software Evolution (IWPSE 1)*, pp. 20-21, 1998.

[293] Oreizy P., Medvidovic N., and Taylor R. N., “Architecture-based runtime software evolution”, In *Proceedings of International Conference on Software Engineering*, pp. 177-186, IEEE Computer Society, 1998.

[294] Paepcke A., Chang C.-C. K., Winograd T., and García-Molina H., “Interoperability for digital libraries worldwide”, In *Communications of the ACM*, volume 41, issue 4, pp. 33-43, 1998.

[295] Palsberg J., “Software Evolution and Integration”, In *ACM Computing Surveys*, volume 28, issue 4es, 1996, http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a200-palsberg/.

[296] Parikh G., *Handbook of Software Maintenance*, ISBN 047-180930-6, Wiley Interscience, 1986.

[297] Parnas D. L., “On the Criteria To Be Used in Decomposing Systems into Modules”, In *Communications of the ACM* , volume 15, issue 12, pp. 1053-1058, 1972.

[298] Parnas D. L., “On a 'buzzword': Hierarchical Structure”, In *Proceedings of IFIP Congress*, pp. 336-339, 1974.

[299] Parnas D. L., “Software Aging”, In *Proceedings of The 16th International Conference on Software Engineering*, pp. 279-287, IEEE Press, 1994.

[300] Patton M. Q., *Qualitative Research & Evaluation Methods* (3rd edition), ISBN 0-7619-1971-6, Sage Publications, 2002.

[301] Paulish D., *Architecture-Centric Software Project Management: A Practical Guide*, SEI Series in Software Engineering, ISBN 0-201-73409-5, Addison-Wesley, 2002.

[302] Perry D. E., “Laws and principles of evolution”, In *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 70-70, IEEE, 2002.

[303] Perry D. E. and Wolf A. L., “Foundations for the study of software architecture”, In *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, pp. 40-52, 1992.

[304] Perry D. E., Siy H. P., and Votta L. G., “Parallel Changes in Large-Scale Software Development: An Observational Case Study”, In *ACM Transactions on Software Engineering and Methodology*, volume 10, issue 3, pp. 308-337, 2001.

[305] Pigoski T. M., *Practical Software Maintenance*, ISBN 0471-17001-1, John Wiley & Sons, 1997.

[306] Pinto H. S. and Martins J. P., “A Methodology for Ontology Integration”, In *Proceedings of International Conference on Knowledge Capture*, pp. 131-138, ACM, 2001.

[307] Pollock J. T., “The Big Issue: Interoperability vs. Integration”, In *eAI Journal*, volume October, 2001, http://www.eaijournal.com/.

[308] Popper K. R., *The Logic of Scientific Discovery*, ISBN 041507892X, Routledge Press, 1959.

[309] Pressman R. S., *Software Engineering — A Practitioner's Approach*, McGraw-Hill International Ltd., 2000.

[310] Ramage M. and Bennett K., “Maintaining maintainability”, In *Proceedings of International Comference on Software Maintenance (ICSM)*, pp. 275-281, IEEE, 1998.

[311] Ramil J. F. and Lehman M. M., "Metrics of Software Evolution as Effort Predictors - A Case Study", In *Proceedings of International Conference on Software Maintenance*, pp. 163-172, IEEE, 2000.

[312] Ramil J. F. and Lehman M. M., "Defining and applying metrics in the context of continuing software evolution", In *Proceedings of Seventh International Software Metrics Symposium (METRICS)*, pp. 199-209, IEEE, 2001.

[313] Robson C., *Real World Research* (2nd edition), ISBN 0-631-21305-8, Blackwell Publishers, 2002.

[314] Roman E., *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*, ISBN 0-471-33229-1, Wiley, 1999.

[315] Royster C., "DoD Strategy on Open Systems and Interoperability", In *StandardView*, volume 4, issue 2, pp. 104-106, 1996.

[316] Rozanski N. and Woods E., *Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives*, ISBN 0-321-11229-6, Addison-Wesley, 2005.

[317] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[318] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object-oriented Modeling and Design*, ISBN 0136300545, Prentice Hall, 1991.

[319] Rumpe B., Schoenmakers M., Radermacher A., and Schurr A., "UML+ROOM as a standard ADL?", In *Proceedings of Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pp. 43-53, 1999.

[320] Russell B., *Our Knowledge of the External World*, ISBN 0415096057, Routledge, 1993.

[321] Rönkkö K., *Making Methods Work in Software Engineering : Method Deployment - as a Social Achievement*, Ph.D. Thesis, Blekinge Institute of Technology, 2005.

[322] Sage A. P. and Lynch C. L., "Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives", In *Systems Engineering*, volume 1, issue 3, pp. 176-227, 1998.

[323] Sametinger J., *Software Engineering with Reusable Components*, ISBN ISBN: 3-540-62695-6, Springer, 1997.

[324] SAP, www.sap.com, *SAP R/3*, www.sap.com, 2003.

[325] Sarma A., Noroozi Z., and van der Hoek A., "Palantír: Raising Awareness among Configuration Management Workspaces" , In *Proceedings of Twenty-Fifth International Conference on Software Engineering*, pp. 444-454, IEEE, 2003.

[326] Sarma A. and van der Hoek A., "Visualizing Parallel Workspace Activities", In *Proceedings of IASTED International Conference on Software Engineering and Applications*, pp. 435-440, IASTED, 2003.

[327] Sartipi K. and Kontogiannis K., "A graph pattern matching approach to software architecture recovery", In *Proceedings of International Conference on Software Maintenance*, pp. 408-419, IEEE, 2001.

[328] Sauer L. D., Clay R. L., and Armstrong R., "Meta-component architecture for software interoperability", In *Proceedings of International Conference on Software Methods and Tools (SMT)*, pp. 75-84, IEEE, 2000.

[329] Schmidt D., Stal M., Rohnert H., and Buschmann F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, Wiley Series in Software Design Patterns, ISBN 0-471-60695-2, John Wiley & Sons Ltd., 2000.

[330] Schwanke R. W., "An intelligent tool for re-engineering software modularity", In *Proceedings of 13th International Conference on Software Engineering*, pp. 83-92, ACM, 1991.

[331] Schwanke R., "GEAR: A Good Enough Architectural Requirements Process", In *Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture*, pp. 57-66, IEEE, 2005.

[332] Seaman C. B., "Qualitative Methods in Empirical Studies of Software Engineering", In *IEEE Transactions on Software Engineering*, volume 25, issue 4, pp. 557-572, 1999.

[333] SEI, *Architecture Description Languages*, URL: http://www.sei.cmu.edu/architecture/adl.html, 2003.

[334] SEI, *The IDEAL Model*, URL: http://www.sei.cmu.edu/ideal/, 2006.

[335] SEI Software Technology Roadmap, *Six Sigma*, URL: http://www.sei.cmu.edu/str/descriptions/sigma6_body.html, 2001.

[336] SEI Software Technology Roadmap, *Architecture Description Languages*, URL: http://www.sei.cmu.edu/str/descriptions/adl_body.html, 2003.

[337] SEI Software Technology Roadmap, *Cyclomatic Complexity*, URL: http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html, 2003.

[338] SEI Software Technology Roadmap, *Halstead Complexity Measures*, URL: http://www.sei.cmu.edu/str/descriptions/halstead_body.html, 2003.

[339] SEI Software Technology Roadmap, *Maintainability Index Technique for Measuring Program Maintainability*, URL: http://www.sei.cmu.edu/str/descriptions/mitmpm.html, 2003.

[340] SEI Software Technology Roadmap, *Three Tier Software Architectures*, URL: http://www.sei.cmu.edu/str/descriptions/threetier_body.html, 2003.

[341] SEI Software Technology Roadmap, *Two Tier Software Architectures*, URL: http://www.sei.cmu.edu/str/descriptions/twotier_body.html, 2003.

[342] SEI Software Technology Roadmap, *COTS and Open Systems--An Overview*, URL: http://www.sei.cmu.edu/str/descriptions/cots.html, 9-6-2004.

[343] SEI Software Technology Roadmap, *Reference Models, Architectures, Implementations--An Overview*, URL: http://www.sei.cmu.edu/str/descriptions/refmodels_body.html, 9-6-2004.

[344] Sewell M. T. and Sewell L. M., *The Software Architect's Profession - An Introduction*, Software Architecture Series, ISBN 0-13-060796-7, Prentice Hall PTR, 2002.

[345] Shanzhen Y., Lizhu Z., Chunxiao X., Qilun L., and Yong Z., "Semantic and interoperable WebGIS", In *Proceedings of the Second International Conference on Web Information Systems Engineering*, pp. 42-47, IEEE, 2001.

[346] Shaw M., "Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status", In *Proceedings of Workshop on Studies of Software Design*, Lecture Notes in Computer Science 1078, Springer-Verlag, 1996.

[347] Shaw M., "The Coming-of-Age of Software Architecture Research", In *Proceedings of 23rd International Conference on Software Engineering (ICSE)*, pp. 657-664, ACM, 2001.

[348] Shaw M., "What Makes Good Research in Software Engineering?", In *International Journal of Software Tools for Technology Transfer*, volume 4, issue 1, pp. 1-7, 2002.

[349] Shaw M. and Clements P., "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", In *Proceedings of The 21st Computer Software and Applications Conference*, 1994.

[350] Shaw M. and Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, ISBN 0-13-182957-2, Prentice-Hall, 1996.

[351] Siegel J., *CORBA 3 Fundamentals and Programming* (2nd edition), ISBN 0471295183, John Wiley & Sons, 2000.

[352] Sommerville I., *Software Engineering* (7th edition), ISBN 0-321-21026-3, Addison-Wesley, 2004.

[353] Springer, *Springer - Academic Journals, Books and Online Media*, URL: http://www.springer.com/, 2006.

[354] Stafford J. and Wallnau K. C., "Component Composition and Integration", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[355] Stanford University, *Stanford Annotated Interoperability Bibliography*, URL: www-diglib.stanford.edu/diglib/pub/interopbib.html, 8-4-2004.

[356] Staples M. and Niazi M., "Experiences Using Systematic Review Guidelines", In *Proceedings of 10th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Keele University, UK, 2006.

[357] Stapleton J., *DSDM - Dynamic Systems Development Method*, ISBN 0-201-17889-3, Pearson Education, 1997.

[358] Stonebraker M. and Hellerstein J. M., "Content Integration for E-Business", In *ACM SIGMOD Record*, volume 30, issue 2, pp. 552-560, 2001.

[359] Strauss A. and Corbin J. M., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (2nd edition), ISBN 0803959400, Sage Publications, 1998.

[360] Svahnberg M. and Bosch J., "Characterizing Evolution in Product Line Architectures", In *Proceedings of 3rd annual IASTED*

International Conference on Software Engineering and Applications, pp. 92-97, IASTED/Acta Press, 1999.

[361] Svahnberg M. and Bosch J., "Issues Concerning Variability in Software Product Lines", In *Proceedings of Software Architectures for Product Families: 7th International Workshop on Database Programming Languages, DBPL'99, Revised Papers (Lecture Notes in Computer Science 1951)*, Springer Verlag, 2000.

[362] Szyperski C., *Component Software - Beyond Object-Oriented Programming*, ISBN 0-201-17888-5, Addison-Wesley, 1998.

[363] Taxén L., *A Framework for the Coordination of Complex Systems' Development*, Ph.D. Thesis, Linköping University, Department of Computer and Information Science, 2003.

[364] Teng W., Pollack N., Serafino G., Chiu L., and Sweatman P., "GIS and Data Interoperability at the NASA Goddard DAAC", In *Proceedings of International Geoscience and Remote Sensing Symposium (IGARSS)*, pp. 1953-1955, IEEE, 2001.

[365] Thai T. and Lam H., *.NET Framework Essentials* (2nd edition), O'Reilly Programming Series, ISBN 0596003021, O'Reilly & Associates, 2002.

[366] The Software Revolution, *The Software Revolution, Inc.*, URL: http://www.softwarerevolution.com/, 2006.

[367] Tichy W., Lukowicz P., Prechelt L., and Heinz E. A., "Experimental evaluation in computer science: A quantitative study", In *Journal of Systems & Software*, volume 28, issue 1, pp. 9-19, 1995.

[368] Toussaint P. J., *Integration of Information Systems : a Study in Requirements Engineering*, Ph.D. Thesis, Rijksuniversiteit Leiden, 1998.

[369] Tu Q. and Godfrey M. W., "The build-time software architecture view", In *Proceedings of International Conference on Software Maintenance*, pp. 398-407, IEEE, 2001.

[370] Tu Q. and Godfrey M. W., "An integrated approach for studying architectural evolution", In *Proceedings of 10th International Workshop on Program Comprehension*, pp. 127-136, 2002.

[371] Tu S., Xu L., Abdelguerfi M., and Ratcliff J. J., "Applications: Achieving Interoperability for Integration of Heterogeneous COTS Geographic Information Systems", In *Proceedings of Tenth ACM*

International Symposium on Advances in Geographic Information Systems, pp. 162-167, ACM, 2002.

[372] Turing A., "On Computable Numbers, with an application to the Entscheidungsproblem", In *Proc.Lond.Math.Soc.*, volume 2, issue 42, pp. 230-265, 1937.

[373] UML, *UML Home Page*, URL: http://www.uml.org/, 2003.

[374] Usabilitynet, *Usabilitynet: usability resources for practitioners and managers*, URL: http://www.usabilitynet.org/home.htm, 2006.

[375] van der Hoek A., Heimbigner D., and Wolf A. L., *Versioned Software Architecture*, 1998.

[376] van der Hoek A., Heimbigner D., and Wolf A. L., *Capturing Architectural Configurability: Variants, Options, and Evolution*, Technical Report CU-CS-895-99, 1999.

[377] van der Hoek A., Mikic-Rakic M., Roshandel R., and Medvidovic N., "Taming Architectural Evolution", In *Proceedings of The Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[378] van der Ven J. S., Jansen A., Avgeriou P., and Hammer D., "Using Architectural Decisions", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, 2006.

[379] van der Westhuizen C. and van der Hoek A., "Understanding and Propagating Architectural Change", In *Proceedings of Third Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3)*, pp. 95-109, Kluwer Academic Publishers, 2002.

[380] van Deursen A., "The Software Evolution Paradox: An Aspect Mining Perspective", In *Proceedings of International ERCIM Workshop on Software Evolution*, 2006.

[381] van Gurp J. and Bosch J., "Design Erosion: Problems & Causes", In *Journal of Systems & Software*, volume 61, issue 2, pp. 105-119, 2002.

[382] van Ommering R., "The Koala Component Model", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[383] van Ommering R., van der Linden F., and Kramer J., "The Koala Component Model for Consumer Electronics Software", In *IEEE Computer*, volume 33, issue 3, pp. 78-85, 2000.

[384] van Vliet H., *Software Engineering : Principles and Practice*, ISBN 0 471 93611 1, John Wiley & Sons, 1993.

[385] Visser U., Stuckenschmidt H., Schuster G., and Vögele T., "Ontologies for Geographic Information Processing", In *Computers & Geosciences*, volume 28, issue 1, pp. 103-117, 2002.

[386] Wall A., *Software Architectures - An Overview*, Department of Computer Engineering, Mälardalen University, 1998.

[387] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.

[388] Wegner P., "Interoperability", In *ACM Computing Surveys*, volume 28, issue 1, 1996.

[389] Welker K. D. and Oman P., "Software Maintainability Metrics Models in Practice", In *Crosstalk - the Journal of Defense Software Engineering*, issue Nov/Dec, 1995.

[390] Whitehead E. J. Jr., "An architectural model for application integration in open hypermedia environments", In *Proceedings of Conference on Hypertext and Hypermedia*, pp. 1-12, ACM Press, 1997.

[391] Wiggerts T. A., "Using clustering algorithms in legacy systems remodularization", In *Proceedings of Fourth Working Conference on Reverse Engineering*, pp. 33-43, IEEE, 1997.

[392] Wikipedia, *Wikipedia, the free encyclopedia*, URL: wikipedia.org, 2006.

[393] Wileden J. C. and Kaplan A., "Software Interoperability: Principles and Practice", In *Proceedings of 21st International Conference on Software Engineering*, pp. 675-676, ACM, 1999.

[394] Wileden J. C., Wolf A. L., Rosenblatt W. R., and Tarr P. L., "Specification Level Interoperability", In *Proceedings of 12th International Conference on Software Engineering (ICSE)*, pp. 74-85, ACM, 1990.

[395] Wohlin C., Runeson P., Höst M., Ohlsson M. C., Regnell B., and Wesslén A., *Experimentation in Software Engineering: An Introduction* (The Kluwer International Series in Software Engineering edition), ISBN 0792386825, Kluwer Academic Publishers, 1999.

[396] Wooldridge M., *Introduction to MultiAgent Systems*, ISBN 047149691X, John Wiley & Sons, 2002.

[397] Wu W. and Kelly T., "Managing Architectural Design Decisions for Safety-Critical Software Systems", In *Proceedings of 2nd International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2006.

[398] WWISA, *Worldwide Institute of Software Architects*, URL: http://www.wwisa.org, 2006.

[399] XML, *XML.org*, URL: http://www.xml.org/, 2004.

[400] Yakimovich D., Bieman J. M., and Basili V. R., "Software Architecture Classification for Estimating the Cost of COTS Integration", In *Proceedings of the 21st International Conference on Software Engineering*, pp. 296-302, ACM, 1999.

[401] Yang J. and Papazoglou M. P., "Interoperation Support for Electronic Business", In *Communications of the ACM* , volume 43, issue 6, pp. 39-47, 2000.

[402] Yau S. S. and Dong N., "Integration in component-based software development using design patterns", In *Proceedings of The 24th Annual International Computer Software and Applications Conference (COMPSAC)*, pp. 369-374, IEEE, 2000.

[403] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

[404] Young P., Chaki N., Berzins V., and Luqi, "Evaluation of Middleware Architectures in Achieving System Interoperability", In *Proceedings of 14th IEEE International Workshop on Rapid Systems Prototyping*, pp. 108-116, IEEE, 2003.

[405] Zachman J. A., "A Framework for Information Systems Architecture", In *IBM Systems Journal*, volume 26, issue 3, 1987.

[406] Zelkowitz M. V. and Wallace D., "Experimental validation in software engineering", In *Information and Software Technology*, volume 39, issue 11, pp. 735-743, 1997.

[407] Zhang C., "Formal Semantic Specification for a Set of UML Diagrams", In *Proceedings of International Conference on Software Engineering Research and Practice*, CSREA Press, 2003.

[408] Zhuo F., Lowther B., Oman P., and Hagemeister J., "Constructing and testing software maintainability assessment models", In

Proceedings of First International Software Metrics Symposium, pp. 61-70, IEEE, 1993.

[409] ZIFA, *Zachman Framework for Enterprise Architecture*, URL: http://www.zifa.com/, 2003.

Paper I

Paper I

This paper is a reprint of:

"Software Systems Integration and Architectural Analysis – A Case Study", Rikard Land, Ivica Crnkovic, *Proceedings of International Conference on Software Maintenance (ICSM)*, Amsterdam, Netherlands, September 2003

The questionnaire form used to collect data from project participants, and the collected data, can be found in Appendix A.

Abstract

Software systems no longer evolve as separate entities but are also integrated with each other. The purpose of integrating software systems can be to increase user-value or to decrease maintenance costs. Different approaches, one of which is software architectural analysis, can be used in the process of integration planning and design.

This paper presents a case study in which three software systems were to be integrated. We show how architectural reasoning was used to design and compare integration alternatives. In particular, four different levels of the integration were discussed (interoperation, a so-called Enterprise Application Integration, an integration based on a common data model, and a full integration). We also show how cost, time to delivery and maintainability of the integrated solution were estimated.

On the basis of the case study, we analyze the advantages and limits of the architectural approach as such and conclude by outlining directions for future research: how to incorporate analysis of cost, time to delivery, and risk in architectural analysis, and how to make architectural analysis more suitable for comparing many aspects of many alternatives during development. Finally we outline the limitations of architectural analysis.

Keywords

Architectural Analysis, Enterprise Application Integration, Information Systems, Legacy Systems, Software Architecture, Software Integration.

1. Introduction

The evolution, migration and integration of existing software (legacy) systems are widespread and a formidable challenge to today's businesses [4,19]. This paper will focus on the *integration* of software systems. Systems need to be integrated for many reasons. In an organization, processes are usually supported by several tools and there is a need for integration of these tools to achieve an integrated and seamless process. Company mergers demand increased interoperability and integration of tools. Such tools can be very diverse with respect to technologies, structures and use and their integration can therefore be very complex, tedious, and time- and effort-consuming. One important question which arises: Is it feasible to integrate these tools and which approach is the best to analyze, design and implement the integration?

[Copyrighted pages 111-130 have been removed, but the original paper can be retrieved from the IEEE or from the author]

Paper II

Paper II

This paper is a reprint of:

"Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", Rikard Land, Ivica Crnkovic, accepted for publication in *Journal of Information and Software Technology*, January 2006

The open-ended interview questions used to collect data are reprinted in Appendix B.

Abstract

As organizations merge or collaborate closely, an important question is how their existing software assets should be handled. If these previously separate organizations are in the same business domain – they might even have been competitors – it is likely that they have developed similar software systems. To rationalize, these existing software assets should be integrated, in the sense that similar features should be implemented only once. The integration can be achieved in different ways. Success of it involves properly managing challenges such as making as well founded decisions as early as possible, maintaining commitment within the organization, managing the complexities of distributed teams, and synchronizing the integration efforts with concurrent evolution of the existing systems.

This paper presents a multiple case study involving nine cases of such in-house integration processes. Based both on positive and negative experiences of the cases, we pinpoint crucial issues to consider early in the process, and suggest a number of process practices.

Keywords

Software integration, software merge, strategic decisions, architectural compatibility

1 Introduction

When organizations merge, or collaborate very closely, they often bring a legacy of in-house developed software systems, systems that address similar problems within the same business. As these systems address similar problems in the same domain, there is usually some overlap in functionality and purpose. Independent of whether the software systems are products or are mainly used in-house, it makes little economic sense to evolve and maintain systems separately. A single implementation combining the functionality of the existing systems would improve the situation both from an economical and maintenance point of view, and also from the point of view of users, marketing and customers. This situation may also occur as systems with initially different purposes are developed in-house (typically by different parts of the organization), and evolved and extended until they partially implement the same basic functionality; the global optimum for the organization as a whole would be to integrate these systems into one, so that there is a single implementation of the same functionality.

[Copyrighted pages 135-192 have been removed, but the original paper can be retrieved from Elsevier or from the author]

Paper III

Paper III

This paper is a reprint of:

"Integration of Software Systems – Process Challenges", Rikard Land, Ivica Crnkovic, Christina Wallin, *Proceedings of Euromicro Conference, Track on Software Process and Product Improvement (SPPI)*, Antalya, Turkey, September 2003

The questionnaire form used to collect data from project participants, and the collected data, can be found in Appendix A.

Abstract

The assumptions, requirements, and goals of integrating existing software systems are different compared to other software activities such as maintenance and development, implying that the integration processes should be different. But where there are similarities, proven processes should be used.

In this paper, we analyze the process used by a recently merged company, with the goal of deciding on an integration approach for three systems. We point out observations that illustrate key elements of such a process, as well as challenges for the future.

Keywords

Software Architecture, Software Evolution, Software Integration, Software Process Improvement.

1. Introduction

Software integration as a special type of software evolution has become more and more important in recent years [7], but brings new challenges and complexities. There are many reasons for software integration; in many cases software integration is a result of company mergers. In this paper we describe such a case, which illustrates the challenges of the decision process involved in deciding the basic principles of the integration on the architectural level.

2. Case Study

Our case study concerns a large North-American industrial enterprise with thousands of employees that acquired a smaller (~800 employees) European company in the same, non-software, business area where software, mainly in-house developed, is used for simulations and management of simulation data, i.e. as tools for development and production of other products. The expected benefits of an integration were increased value for users (more functionality and all related data collected in the same system) as well as more efficient use of software development and maintenance resources. The first task was to make a decision on an architecture to choose for the integrated system. The present paper describes this decision process.

Paper IV

Paper IV

This paper is a reprint of:

"Software In-House Integration – Quantified Experiences from Industry", Rikard Land, Peter Thilenius, Stig Larsson, Ivica Crnkovic, *Proceedings of Euromicro Conference Software Engineering and Advanced Applications, Track on Software Process and Product Improvement (SPPI)*, Cavtat, Croatia, August-September 2006

The questionnaire form used for data collection is reprinted in Appendix D and the collected data in Appendix E.

Abstract

When an organization faces new types of collaboration, for example after a company merger, there is a need to consolidate the existing in-house developed software. There are many high-level strategic decisions to be made, which should be based on as good foundation as possible, while these decisions must be made rapidly. Also, one must employ feasible processes and practices in order to get the two previously separate organizations to work towards a common goal. In order to study this topic, we previously performed an explorative and qualitative multiple case study, where we identified a number of suggested practices as well as other concerns to take into account. This paper presents a follow-up study, which aims at validating and quantifying these previous findings. This study includes a questionnaire distributed to in-house integration projects, aiming at validation of earlier findings. We compare the data to our previous conclusions, present observations on retirement of the existing systems and on the technical similarities of the existing systems. We also present some practices considered important but often neglected.

1. Introduction

When organizations merge, or collaborate very closely, they often bring a legacy of in-house developed software systems, systems that address similar problems within the same business. As these systems address similar problems in the same domain, there is usually some overlap in functionality and purpose. It makes little economic sense to evolve and maintain these systems separately (this is true for any kind of system built internally, independent of whether they are core products offered to the market or are internally built tools mainly used in-house). A single coherent system would be ideal. This situation may also occur as systems are independently developed by different parts of the same organization; as they grow a point will be reached where there is too much overlap, and should be integrated. This paper presents the results of a questionnaire survey designed to study this topic, which we have labelled “in-house integration”.

The questionnaire is based on earlier observations from an explorative qualitative multiple case study [29]. This previous study consisted of nine cases from six organizations. The main data source was interviews, but in several cases, we also had access to certain documentation. The previous material was analyzed from several points of view [16]:

Paper V

Paper V

This paper is a reprint of:

"Merging In-House Developed Software Systems – A Method for Exploring Alternatives", Rikard Land, Jan Carlson, Stig Larsson, Ivica Crnkovic, Proceedings of the *International Conference on the Quality of Software Architecture*, Västerås, Sweden, June 2006

The open-ended interview questions used to collect case study data are reprinted in Appendix C.

Abstract

An increasing form of software evolution is software merge – when two or more software systems are being merged. The reason may be to achieve new integrated functions, but also remove duplication of services, code, data, etc. This situation might occur as systems are evolved in-house, or after a company acquisition or merger. One potential solution is to merge the systems by taking components from the two (or more) existing systems and assemble them into an existing system. The paper presents a method for exploring merge alternatives at the architectural level, and evaluates the implications in terms of system features and quality, and the effort needed for the implementation. The method builds on previous observations from several case studies. The method includes well-defined core model with a layer of heuristics in terms of a loosely defined process on top. As an illustration of the method usage a case study is discussed using the method.

1. Introduction

When organizations merge, or collaborate very closely, they often bring a legacy of in-house developed software systems. Often these systems address similar problems within the same business and there is usually some overlap in functionality and purpose. A new system, combining the functionality of the existing systems, would improve the situation from an economical and maintenance point of view, as well as from the point of view of users, marketing and customers. During a previous study involving nine cases of such in-house integration [10], we saw some drastic strategies, involving retiring (some of) the existing systems and reusing some parts, or only reutilizing knowledge and building a new system from scratch. We also saw another strategy of resolving this situation, which is the focus of the present paper: to merge the systems, by reassembling various parts from several existing system into a new system. From many points of view, this is a desirable solution, but based on previous research this is typically very difficult and is not so common in practice; there seem to be some prerequisites for this to be possible and feasible [10].

There is a need to relatively fast and accurately find and evaluate merge solutions, and our starting point to address this need has been the following previous observations [10]:

1. Similar high-level structures seem to be a prerequisite for merge. Thus, if the structures of the existing systems are not similar, a merge seems in practice unfeasible.

2. A development-time view of the system is a simple and powerful system representation, which lends itself to reasoning about project characteristics, such as division of work and effort estimations.
3. A suggested beneficial practice is to assemble the architects of the existing systems in a meeting early in the process, where various solutions are outlined and discussed. During this type of meeting, many alternatives are partly developed and evaluated until (hopefully) one or a few high-level alternatives are fully elaborated.
4. The merge will probably take a long time. To sustain commitment within the organization, and avoid too much of parallel development, there is a need to perform an evolutionary merge with stepwise deliveries. To enable this, the existing systems should be delivered separately, sharing more and more parts until the systems are identical.

This paper presents a systematic method for exploring merge alternatives, which takes these observations into account: by 1) assuming similar high-level structures, 2) utilizing static views of the systems, 3) being simple enough to be able to learn and use during the architects' meetings, and 4) by focusing not only on an ideal future system but also stepwise deliveries of the existing systems. The information gathered from nine case studies was generalized into the method presented in this paper. To refine the method, we made further interviews with participants in one of the previous cases, which implemented the merge strategy most clearly.

The rest of the paper is organized as follows. We define the method in Section 2 and discuss it by means of an example in Section 3. Section 4 discusses important observations from the case and argues for some general advices based on this. Section 5 surveys related work. Section 6 summarizes and concludes the paper and outlines future work.

2. Software Merge Exploration Method

Our software merge exploration method consists of two parts: (i) a model, i.e., a set of formal concepts and definitions, and (ii) a process, i.e., a set of human activities that utilizes the model. The model is designed to be simple but should reflect reality as much as possible, and the process describes higher-level reasoning and heuristics that are suggested as useful practices.

To help explaining the method, we start with a simple example in Section 2.1, followed by a description of the method's underlying model (Section 2.2) and the suggested process (Section 2.3).

2.1 An Explanatory Example

Figure 1a shows two simple music sequencer software systems structured according to the "Model-View-Controller" pattern [2]. The recorded music would be the model, which can be viewed as a note score or as a list of detailed events, and controlled by mouse clicks or by playing a keyboard.

The method uses the module view [3,5] (or development view [8]), which describes modules and "use" dependencies between them. Parnas defined the "use" dependency so that module α is said to use module β if module α relies on the correct behavior of β to accomplish its task [14].

In our method, the term *module* refers to an encapsulation of a particular functionality, purpose or responsibility on an abstract level. A concrete implementation of this functionality is called a *module instance*. In the example, both systems have a EventView module, meaning that both systems provide this particular type of functionality (e.g., a note score view of the music). The details are probably different in the two systems, though, since the functionality is provided by different concrete implementations (the module instances $\mathsf{EventView}_{\mathsf{A}}$ and $\mathsf{EventView}_{\mathsf{B}}$, respectively). The method is not restricted to module instances that are present in the existing systems but also those that are possible in a future system; such new module instances could be either a planned implementation (e.g., $\mathsf{EventView}_{\mathsf{new\_impl}}$), an already existing module to be reused in-house from some other program (e.g., $\mathsf{EventView}_{\mathsf{pgm\_name}}$), or an open source or commercial component ($\mathsf{EventView}_{\mathsf{component\_name}}$).

2.2 The Model

Our proposed method builds on a model consisting of three parts: a set of model elements, a definition of inconsistency in terms of the systems' structures, and a set of permissible user operations.

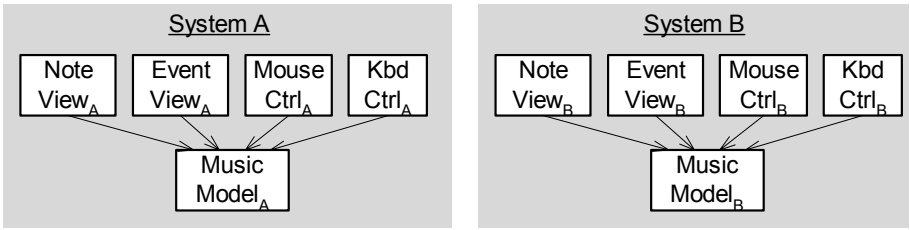
2.2.1 Concepts and Notation

The following concepts are used in the model:

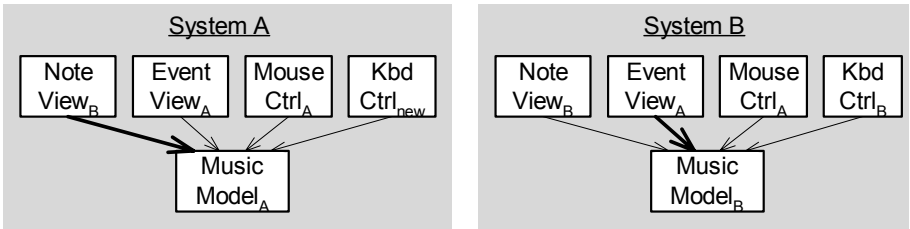
- We assume there are two or more existing *systems*, (named with capital letters, and parameterized by X, Y, etc.).
- A *module* represents a conceptual system part with a specific purpose (e.g., EventView in Figure 1). Modules are designated with capital first letter; in the general case we use Greek letters α and β.

- A *module instance* represents a realization of a module. It is denoted α_X where α is a module and X is either an existing system (as in $\mathsf{EventView}_A$) or an indication that the module is new to the systems (as in $\mathsf{EventView}_{pgm\_name}$ or $\mathsf{EventView}_{component\_name}$).
- A *"use" dependency* (or *dependency* for short) from module instance α_X to module instance β_Y means that α_X relies on the correct behavior of β_Y to accomplish its task. We use the textual notation $\alpha_X \rightarrow \beta_Y$ to represent this.
- A *dependency graph* captures the structure of a system. It is a directed graph where each node in the graph represents a module instance and the edges (arrows) represent use dependencies. In Figure 1a, we have for example the dependencies $\mathsf{NoteView}_A \rightarrow \mathsf{MusicModel}_A$ and $\mathsf{MouseCtrl}_B \rightarrow \mathsf{MusicModel}_B$.
- An *adaptation* describes that a modification is made to α_X in order for it to be compatible, or *consistent* with β_Y, and is denoted $\langle \alpha_X, \beta_Y \rangle$ (see 2.2.2 below).
- A *scenario* consists of a dependency graph for each existing system and a single set of adaptations.

a) Initial state



b) State after some changes have been made to the systems



Adaptation Set: <$\mathsf{KbdCtrl}_{new}$, $\mathsf{MusicModel}_A$> <$\mathsf{MusicModel}_B$, $\mathsf{MouseCtrl}_A$>

Figure 1. Two example systems with the same structure being merged.

2.2.2 Inconsistency

A dependency from α_X to β_Y can be *inconsistent*, meaning that β_Y cannot be used by α_X. Trivially, the dependency between two module instances from the same system is consistent without further adaptation. For the dependency between two modules from different systems we cannot say whether they are consistent or not. Most probably they are inconsistent, which has to be resolved by some kind of adaptation if we want to use them together in a new system. The actual adaptations made could in practice be of many kinds: some wrapping or bridging code, or modifications of individual lines of code; see further discussion in 4.1.

Formally, a dependency $\alpha_X \rightarrow \beta_Y$ is *consistent* if $X = Y$ or if the adaptation set contains $\langle \alpha_X, \beta_Y \rangle$ or $\langle \beta_Y, \alpha_X \rangle$. Otherwise, the dependency is *inconsistent*. A dependency graph is consistent if all dependencies are consistent; otherwise it is inconsistent. A scenario is consistent if all dependency graphs are consistent; otherwise it is inconsistent.

Example: The scenario in Figure 1b is inconsistent, because of the inconsistent dependencies from $\mathsf{NoteView}_B$ to $\mathsf{MusicModel}_A$ (in System A) and from $\mathsf{EventView}_A$ to $\mathsf{MusicModel}_B$ (in System B). The dependencies from $\mathsf{KbdCtrl}_{new}$ to $\mathsf{MusicModel}_A$ (in System A) and from $\mathsf{MouseCtrl}_A$ to $\mathsf{MusicModel}_B$ (in System B) on the other hand are consistent, since there are adaptations $\langle \mathsf{KbdCtrl}_{new}, \mathsf{MusicModel}_A \rangle$ and $\langle \mathsf{MusicModel}_B, \mathsf{MouseCtrl}_A \rangle$ representing that $\mathsf{KbdCtrl}_{new}$ and $\mathsf{MusicModel}_B$ have been modified to be consistent with $\mathsf{MusicModel}_A$ and $\mathsf{MouseCtrl}_A$ respectively.

2.2.3 Scenario Operations

The following operations can be performed on a scenario:

1. Add an adaptation to the adaptation set.
2. Remove an adaptation from the adaptation set.
3. Add the module instance α_X to one of the dependency graphs, if there exists an α_Y in the graph. Additionally, for each module β, such that there is a dependency $\alpha_Y \rightarrow \beta_Z$ in the graph, a dependency $\alpha_X \rightarrow \beta_W$ must be added for some β_W in the graph.
4. Add the dependency $\alpha_X \rightarrow \beta_W$ if there exist a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.
5. Remove the dependency $\alpha_X \rightarrow \beta_W$ if there exists a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.

6. Remove the module instance α_X from one of the dependency graphs, if there are no edges to α_X in the graph, and if the graph contains another module instance α_Y (i.e., with $X \neq Y$).

Note that these operations never change the participating modules of the graphs (if there is an α_X in the initial systems, they will always contain some α_Y). Similarly, dependencies between modules are also preserved Note also that we allow two or more instances for the same module in a system; when this could be suitable for a real system is discussed in 4.2.

2.3 The Process

The suggested process consists of two phases, the first consisting of two simple preparatory activities (P-I and P-II), and the second being recursive and exploratory (E-I – E-IV).

The scope of the method is within an early meeting of architects, where they (among other tasks) outline various merge solutions. To be able to evaluate various alternatives, some evaluation criteria should be provided by management, product owners, or similar stakeholders. Such criteria can include quality attributes for the system, but also considerations regarding development parameters such as cost and time limits. Other boundary conditions are the strategy for the future architecture and anticipated changes in the development organization. Depending on the circumstances, evaluation criteria and boundary conditions could be renegotiated to some extent, once concrete alternatives are developed.

2.3.1 Preparatory Phase

The *Preparatory* phase consists of two activities:

Activity P-I: Describe Existing Systems. First, the dependency graphs of the existing systems must be prepared, and common modules must be identified. These graphs could be found in existing models or documentation, or extracted by reverse engineering methods, or simply created by the architects themselves.

Activity P-II: Describe Desired Future Architecture. The dependency graph of the future system has the same structure, in terms of modules, as the existing systems. For some modules it may be imperative to use some specific module instance (e.g., α_X because it has richer functionality than α_Y, or a new implementation α_{new} because there have been quality problems with the existing α_X and α_Y). For other modules, α_X might be preferred over α_Y,

but the final choice will also depend on other implications of the choice, which is not known until different alternatives are explored. The result of this activity is an outline of a desired future system, with some annotations, that serve as a guide during the exploratory phase. This should include some quality goals for the system as a whole.

2.3.2 Exploratory Phase

The result of the preparatory phase is a single scenario corresponding to the structure and module instances of the existing systems. The exploratory phase can then be described in terms of four activities: E-I "Introduce Desired Changes", E-II "Resolve Inconsistencies", E-III "Branch Scenarios", and E-IV "Evaluate Scenarios".

The order between them is not pre-determined; any activity could be performed after any of the others. They are however not completely arbitrary: early in the process, there will be an emphasis on activity E-I, where *desired changes* are introduced. These changes will lead to *inconsistencies that need to be resolved* in activity E-II. As the exploration continues, one will need to *branch scenarios* in order to explore different choices; this is done in activity E-III. One also wants to continually *evaluate the scenarios* and compare them, which is done in activity E-IV. Towards the end when there are a number of consistent scenarios there will be an emphasis on evaluating these deliveries of the existing systems. For all these activities, decisions should be described so they are motivated by, and traceable to, the specified evaluation criteria and boundary conditions. These activities describe high-level operations that are often useful, but nothing prohibits the user from carrying out any of the primitive operations defined above at any time.

Activity E-I: Introduce Desired Changes. Some module instances, desired in the future system, should be introduced into the existing systems. In some cases, it is imperative where to start (as described for activity P-II); the choice may e.g., depend on the local priorities for each system (e.g., "we need to improve the MusicModel of system A"), and/or some strategic considerations concerning how to make the envisioned merge succeed (e.g., "the MusicModel should be made a common module as soon as possible").

Activity E-II: Resolve Inconsistencies. As modules are exchanged in the graphs, dependencies $\alpha_X \rightarrow \beta_Y$ might become inconsistent. There are several ways of resolving these inconsistencies:

- Either of the two module instances could be modified to be consistent with the interface of the other. In the model, this means adding an

adaptation to the adaptation set. In the example of Figure 1b, the inconsistency between $\mathsf{NoteView}_B$ and $\mathsf{MusicModel}_A$ in System A can be solved by adding either of the adaptations $\langle \mathsf{NoteView}_B, \mathsf{MusicModel}_A \rangle$ or $\langle \mathsf{MusicModel}_A, \mathsf{NoteView}_B \rangle$ to the adaptation set. (Different types of possible modifications in practice are discussed in Section 4.1.)

- Either of the two module instances could be exchanged for another. There are several variations on this:
  - A module instance is chosen so that the new pair of components is already consistent. This means that α_X is exchanged either for α_Y (which is consistent with β_Y as they come from the same system Y) or for some other α_Z for which there is an adaptation $\langle \alpha_Z, \beta_Y \rangle$ or $\langle \beta_Y, \alpha_Z \rangle$. Alternatively, β_Y is exchanged for β_X or some other β_Z for which there is an adaptation $\langle \beta_Z, \alpha_X \rangle$ or $\langle \alpha_X , \beta_Z \rangle$. In the example of Figure 1b, $\mathsf{MusicModel}_A$ could be replaced by $\mathsf{MusicModel}_B$ to resolve the inconsistent dependency $\mathsf{NoteView}_B \rightarrow \mathsf{MusicModel}_A$ in System A.
  - A module instance is chosen that did not exist in either of the previous systems. This could be either of:
    i) a module reused in-house from some other program (which would come with an adaptation cost),
    ii) a planned or hypothesized new development (which would have an implementation cost, but low or no adaptation cost), or
    iii) an open source or commercial component (which involves acquirement costs as well as adaptation costs, which one would like to keep separate).
- One more module instance could be introduced for one of the modules, to exist in parallel with the existing; the new module instance would be chosen so that it already is consistent with the instance of the other module (as described for exchanging components). The previous example in Figure 1a and b is too simple to illustrate the need for this, but in Section 4 the industrial case will illustrate when this might be needed and feasible. Coexisting modules are also further discussed in Section 4.1.

Some introduced changes will cause new inconsistencies, that need to be resolved (i.e., this activity need to be performed iteratively).

Activity E-III: Branch Scenarios. As a scenario is evolved by applying the operations to it (most often according to either of the high-level approaches of activities E-I and E-II), there will be occasions where it is desired to explore two or more different choices in parallel. For example, several of the resolutions suggested in activity E-II might make intuitive sense, and both choices should be explored. It is then possible to copy the scenario, and treat the two copies as branches of the same tree, having some choices in common but also some different choices.

Activity E-IV: Evaluate Scenarios. As scenarios evolve, they need to be evaluated in order to decide which branches to evolve further and which to abandon. Towards the end of the process, one will also want to evaluate the final alternatives more thoroughly, and compare them – both with each other and with the pre-specified evaluation criteria and boundary conditions (which might at this point be reconsidered to some extent). The actual state of the systems must be evaluated, i.e., the actually chosen module instances plus the modifications to reduce inconsistencies). Do the systems contain many shared modules? Are the chosen modules the ones desired for the future system (richest functionality, highest quality, etc.)? Can the system as a whole be expected to meet its quality goals?

2.3.3 Accumulating Information

As these activities are carried out, there is some information that should be stored for use in later activities. As operations are performed, information is accumulated. Although this information is created as part of an operation within a specific scenario, the information can be used in all other scenarios; this idea would be particularly useful when implemented in a tool. We envision that any particular project or tool would define its own formats and types of information; in the following we give some suggestions of such useful information and how it would be used.

Throughout the exploratory activities, it would be useful to have some ranking of modules readily available, such as "$\mathsf{EventView}_A$ is preferred over $\mathsf{EventView}_B$ because it has higher quality". A tool could use this information to color the chosen modules to show how well the outlined alternatives fit the desired future system.

For activity E-II "Resolve Inconsistencies", it would be useful to have information about e.g., which module could or could not coexist in parallel. Also, some information should be stored that is related to how the inconsistencies are solved. There should at least be a short textual description of what an adaptation means in practice. Other useful

information would be the efforts and costs associated with each acquirement and adaptation; if this information is collected by a tool, it becomes possible to extract a list of actions required per scenario, including the textual descriptions of adaptations and effort estimates. It is also possible to reason about how much of the efforts required that are "wasted", that is: is most of the effort related to modifications that actually lead towards the desired future system, or is much effort required to make modules fit only for the next delivery and then discarded? The evaluation criteria and boundary conditions mentioned in Section 2.2 could also be used by a tool to aid or guide the evaluation in the activity E-IV.

3. An Industrial Case Study

In a previous multiple case study on the topic of in-house integration, the nine cases in six organizations had implemented different integration solutions [10]. We returned to the one case that had clearly chosen the merge strategy and successfully implemented it (although it is not formally released yet); in previous publications this case is labelled "case F2". The fact that this was one case out of nine indicates that the prerequisites for a merge are not always fulfilled, but also that they are not unrealistic (two more cases involved reusing parts from several existing systems in a way that could be described as a merge). To motivate the applicability of the proposed method, this section describes the events of an industrial case and places them in the context of our method.

3.1 Research Method

This part of the research is thus a single case study [17]. Our sources of information have been face-to-face interviews with the three main developers on the US side (there is no title "architect" within the company) and the two main developers on the Swedish side, as well as the high-level documentation of the Swedish system. All discussion questions and answers are published together with more details on the study's design in a technical report [9].

Although the reasoning of the case follows the method closely, the case also demonstrates some inefficiency due to not exploring the technical implications of the merge fully beforehand. It therefore supports the idea of the method being employed to analyze and explore merge alternatives early,

before committing to a particular strategy for the in-house integration (merge or some other strategy).

3.2 The Case

The organization in the case is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. To support the core business, computer simulations are conducted. Both sites have developed software for simulating 3D physics, containing state-of-the-art physics models, many of the models also developed in-house.

As the results are used for real-world decisions potentially affecting the environment and human lives, the simulation results must be accurate (i.e., the output must correspond closely to reality). As the simulations are carried out off-line and the users are physics specialists, many other runtime quality properties of the simulation programs are not crucial, such as reliability (if the program crashes for a certain input, the bug is located and removed), user-friendliness, performance, or portability. On the other side, the accuracy of the results are crucial.

Both systems are written in Fortran and consist of several hundreds of thousands lines of code, and the staff responsible for evolving these simulators are the interviewees, i.e., less than a handful on each site. There was a strategic decision to integrate or merge the systems in the long term. This should be done through cooperation whenever possible, rather than as a separate up-front project.

The rest of this section describes the events of the case in terms of the proposed activities of the method. It should be noted that although the interviewees met in a small group to discuss alternatives, they did not follow the proposed method strictly (which is natural, as the method has been formulated after, and partly influenced by, these events).

Activity P-I: Describe Existing Systems. Both existing systems are written in the same programming language (Fortran), and it was realized early that the two systems have very similar structure, see Figure 2a). There is a main program (Main) invoking a number of physics modules (PX, PY, PZ, …) at appropriate times, within two main loops. Before any calculations, an initialization module (Init) reads data from input files and the internal data structures (DS) are initialized. The physics modeled is complex, leading to complex interactions where the solution of one module affects others in a non-hierarchical manner. After the physics calculations are finished, a file

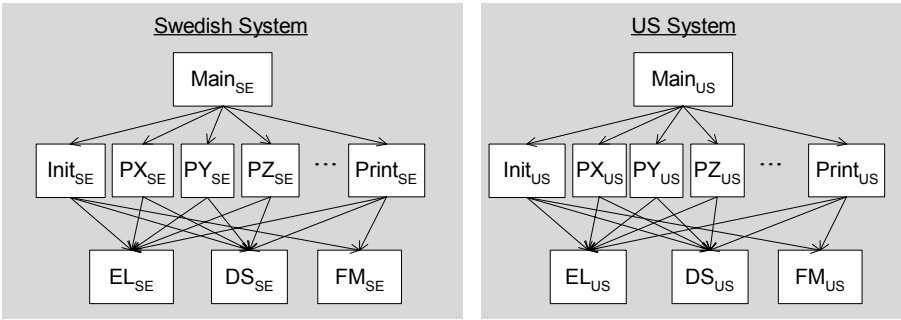
management module (FM) is invoked, which collects and prints the results to file. All these modules use a common error handling and logging library (EL), and share the same data structures (DS). A merge seemed plausible also thanks to the similarities of the data models; the two programs model the same reality in similar ways.

Activity P-II: Describe Desired Future Architecture. The starting point was to develop a common module for one particular aspect of the physics (PX_{new}), as both sides had experienced some limitations of their respective current physics models. Now being in the same company, it was imperative that they would join efforts and develop a new module that would be common to both programs; this project received some extra integration funding. Independent of the integration efforts, there was a common wish on both sides to take advantage of newer Fortran constructs to improve encapsulation and enforce stronger static checks.

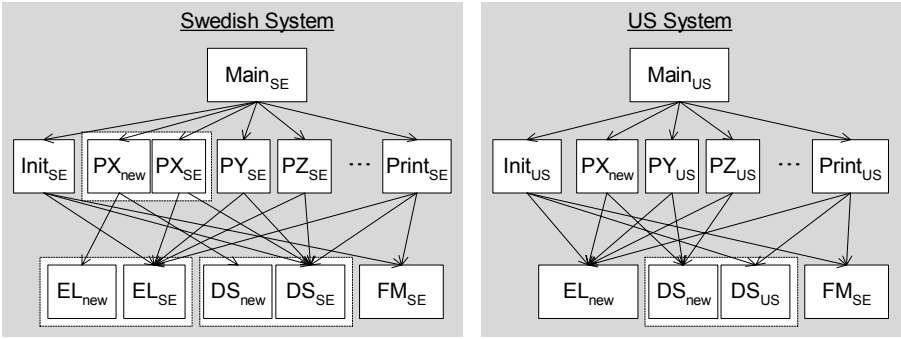
Activity E-I: Introduce Desired Changes. As said, the starting point for integration was the module PX. Both sides wanted a fundamentally new physics model, so the implementation was also completely new (no reuse), written by one of the Swedish developers. The two systems also used different formats for input and output files, managed by file handling modules (FM_{SE} and FM_{US}). The US system chose to incorporate the Swedish module for this, which has required some changes to the modules using the file handling module.

Activity E-II: Resolve Inconsistencies. The PX module of both systems accesses large data structures (DS) in global memory, shared with the other physics modules. An approach was tried where adapters were introduced between a commonly defined interface and the old implementations, but was abandoned as this solution became too complex. Instead, a new implementation of data structures was introduced. This was partially chosen because it gave the opportunity to use newer Fortran constructs which made the code more structured, and it enabled some encapsulation and access control as well as stronger type checking than before.

a) Initial state



b) Current state



Adaptation set: <$Main_{SE}$, PX_{new}> <PX_{new}, EL_{new}> <PX_{new}, DS_{new}> <$Main_{US}$, PX_{new}> <PY_{US}, EL_{new}> <PY_{US}, DS_{new}> <PZ_{US}, EL_{new}> <PZ_{US}, DS_{new}>

c) Future System

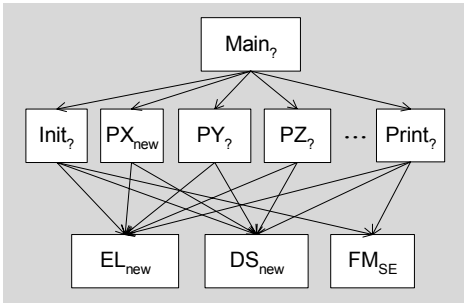


Figure 2: The current status of the systems of the case.

This led to new inconsistencies that needed to be resolved. In the US system, six man-months were spent on modifying the existing code to use the new data structures. The initialization and printout modules remained untouched however; instead a solution was chosen where data is moved from the old structures (DS_{SE} and DS_{US}) to the new (DS_{new}) after the initialization module has populated the old structures, and data is moved back to the old structures before the printout module executes. In the Swedish system, only the parts of the data structures that are used by the PX module are utilized, the other parts of the program uses the old structures; the few data that are used both by the PX module and others had to be handled separately.

The existing libraries for error handling and logging (EL) would also need some improvements in the future. Instead of implementing the new PX module to fit the old EL module, a new EL module was implemented. The new PX module was built to use the new EL module, but the developers saw no major problems to let the old EL module continue to be used by other modules (otherwise there would be an undesirable ripple effect). However, for each internal shipment of the PX module, the US staff commented away the calls to the EL library; this was the fastest way to make it fit. In the short term this was perfectly sensible, since the next US release would only be used for validating the new model together with the old system. However, spending time commenting away code was an inefficient way of working, and eventually the US site incorporated the EL library and modified all other modules to use it; this was not too difficult as it basically involved replacing certain subroutine calls with others. In the Swedish system, the new EL library was used by the new PX module, while the existing EL module was used in parallel, to avoid modifying other modules that used it. Having two parallel EL libraries was not considered a major quality risk in the short run.

Modifying the main loop of each system, to make it call the new PX module instead of the old, was trivial. In the Swedish system there will be a startup switch for some years to come, allowing users to choose between the old and the new PX module for each execution. This is useful for validation of PX_{new} and is presented as a feature for customers.

E-III Branch Scenarios. As we are describing the actual sequence of events, this activity cannot be reported as such, although different alternatives were certainly discussed – and even attempted and abandoned, as for the data structure adapters.

E-IV Evaluate Scenarios. This activity is also difficult to isolate after the fact, as we have no available reports on considerations made. It appears as functionality was a much more important factor than non-functional (quality)

attributes at the module level. At system level, concerns about development time qualities (e.g., discussions about parallel module instances and the impact on maintenance) seem to have been discussed more than runtime qualities (possibly because runtime qualities in this case are not crucial).

Figure 2 shows the initial and current state of the systems, as well as the desired outlined future system. (It is still discussed whether to reuse the module from either of the systems or create a new implementation, hence the question marks).

4. Discussion

This section discusses various considerations to be made during the exploration and evaluation, as highlighted by the case.

4.1 Coexisting Modules

To resolve an inconsistency between two module instances, there is the option of allowing two module instances (operation 2). Replacing the module completely will have cascading effects on the consistencies for all edges connected to it (both "used-by" and "using"), so having several instances has the least direct impact in the model (potentially the least modification efforts). However, it is not always feasible in practice to allow two implementations with the same purpose. The installation and runtime costs associated with having several modules for the same task might be prohibiting if resources are scarce. It might also be fundamentally assumed that there is only one single instance responsible for a certain functionality, e.g., for managing central resources. Examples could be thread creation and allocation, access control to various resources (hardware or software), security, etc. Finally, at development time, coexisting components violates the conceptual integrity of the system, and results in a larger code base and a larger number of interfaces to keep consistent during further evolution and maintenance. From this point of view, coexisting modules might be allowed as a temporary solution for an intermediate delivery, while planning for a future system with a single instance of each module (as in the case for modules EL and DS). However, the case also illustrates how the ability to choose either of the two modules for each new execution was considered useful (PX_{SE} and PX_{new} in the Swedish system).

We can see the following types of relationships between two particular module instances of the same module:

- **Arbitrary usage.** Any of the two parallel modules may be invoked at any time. This seems applicable for library type modules, i.e., modules that retains no state but only performs some action and returns, as the EL module in the case.
- **Alternating usage.** If arbitrary usage cannot be allowed, it might be possible to define some rules for synchronization that will allow both modules to exist in the system. In the case, we saw accesses to old and new data structures in a pre-defined order, which required some means of synchronizing data at the appropriate points in time. One could also imagine other, more dynamic types of synchronization mechanisms useful for other types of systems: a rule stating which module to be called depending on the current mode of the system, or two parallel processes that are synchronized via some shared variables. (Although these kinds of solutions could be seen as a new module, the current version of the method only allows this to be specified as text associated to an adaptation.)
- **Initial choice.** The services of the modules may be infeasible to share between two modules, even over time. Someone will need to select which module instance to use, e.g., at compile time by means of compilation switches, or with an initialization parameter provided by the user at run-time. This was the case for the PX_{SE} and PX_{new} modules in the Swedish system.

The last two types of relationships requires some principle decision and rules at the system (architectural) level, while the signifying feature of the first is that the correct overall behaviour of the program is totally independent of which module instance is used at any particular time.

4.2 Similarity of Systems

As described in 2.1, the model requires that the structures of the existing systems are identical, which may seem a rather strong assumption. It is motivated by the following three arguments [10]:

- The previous multiple case study mentioned in Section 3.1 strongly suggests that similar structures is a prerequisite for merge to make sense in practice. That means that if the structures are dissimilar, practice has shown that some other strategy will very likely be more feasible (e.g.,

involving the retirement of some systems). Consequently, there is little motivation to devise a method that covers also this situation.

- We also observed that it is not so unlikely that systems in the same domain, built during the same era, indeed have similar structures.
- If the structures are not very similar at a detailed level, it might be possible to find a higher level of abstraction where the systems are similar.

A common type of difference, that should not pose large difficulties in practice, is if some modules and dependencies are similar, and the systems have some modules that are only extensions to a common architecture. For example, in the example system one of the systems could have an additional View module (say, a piano roll visualization of the music); in the industrial case we could imagine one of the systems to have a module modeling one more aspect of physics (PW) than the other. However, a simple workaround solution in the current version of the method is to introduce virtual module instances, i.e., modules that do not exist in the real system (which are of course not desired in the future system).

5. Related Work

There is much literature to be found on the topic of *software integration*. Three major fields of software integration are component-based software [16], open systems [13], and Enterprise Application Integration, EAI [15]. However, we have found no existing literature that directly addresses the context of the present research: integration or merge of software *controlled and owned within an organization*. These existing fields address somewhat different problems than ours, as these fields concern components or systems *complementing* each other rather than systems that *overlap* functionally. Also, it is typically assumed that components or systems are acquired from third parties and that modifying them is not an option, a constraint that does not apply to the in-house situation. *Software reuse* typically assumes that components are initially built to be reused in various contexts, as COTS components or as a reuse program implemented throughout an organization [7], but in our context the system components were likely not being built with reuse in mind.

It is commonly expressed that a software architecture should be documented and described according to different views [3,5,6,8]. One frequently proposed view is the module view [3,5] (or development view [8]),

describing development abstractions such as layers and modules and their relationships. The dependencies between the development time artifacts were first defined by Parnas [14] and are during ordinary software evolution the natural tool to understand how modifications made to one component propagate to other.

The notion of "architectural mismatch" is well known, meaning the many types of incompatibilities that may occur when assembling components built under different assumptions and using different technologies [4]. There are some methods for automatically merging software, mainly source code [1], not least in the context of configuration management systems [12]. However, these approaches are unfeasible for merging large systems with complex requirements, functionality, quality, and stakeholder interests. The abstraction level must be higher.

6. Conclusions and Future Work

The problem of integrating and merging large complex software systems owned in-house is essentially unexplored. The method presented in this paper addresses the problem of rapidly outlining various merge alternatives, i.e., exploring how modules could be reused across existing systems to enable an evolutionary merge. The method makes visible various merge alternatives and enables reasoning about the resulting functionality of the merged system as well as about the quality attributes of interest (including both development time and runtime qualities).

The method consists of a formal model with a loosely defined heuristics-based process on top. The goal has been to keep the underlying model as simple as possible while being powerful enough to capture the events of a real industrial case. One of the main drivers during its development has been simplicity, envisioned to be used as a decision support tool at a meeting early in the integration process, with architects of the existing systems. As such, it allows rapid exploration of multiple scenarios in parallel. We have chosen the simplest possible representation of structure, the module view. For simplicity, the method in its current version mandates that the systems have identical structures. This assumption we have shown is not unreasonable but can also be worked around for minor discrepancies. The method is designed so that stepwise deliveries of the existing systems are made, sharing more and more modules, to enable a true evolutionary merge.

Assisted by a tool, it would be possible to conveniently record information concerning all decisions made during the exploration, for later processing and presentation, thus giving an advantage over only paper and pen. We are implementing such a tool, which already exist as a prototype [11]. It displays the graphs of the systems, allows user-friendly operations, highlights inconsistencies with colors, and is highly interactive to support the explorative process suggested. The information collected, in the form of short text descriptions and effort estimations, enables reasoning about subsequent implementation activities. For example, how much effort is the minimum for a first delivery where some module is shared? What parts of a stepwise delivery are only intermediate, and how much effort is thus wasted in the long term?

There are several directions for extending the method: First, understanding and bridging differences in existing data models and technology frameworks of the existing systems is crucial for success and should be part of a merge method. Second, the model could be extended to allow a certain amount of structural differences between systems. Third, the module view is intended to reveal only static dependencies, but other types of relationships are arguably important to consider in reality. Therefore, we intend to investigate how the method can be extended to include more powerful languages, including e.g., different dependency types and different adaptation types, and extended also to other views.

6.1 Acknowledgements

We would like to thank all interviewees and their organization for sharing their experiences and allowing us to publish them. Also thanks to Laurens Blankers for previous collaboration that has led to the present paper, and for our discussions on architectural compatibility.

7. References

[1] Berzins V., “Software merge: semantics of combining changes to programs”, In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16, issue 6, pp. 1875-1903, 1994.

[2] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, John Wiley & Sons, 1996.

[3] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[4] Garlan D., Allen R., and Ockerbloom J., "Architectural Mismatch: Why Reuse is so Hard", In *IEEE Software*, volume 12, issue 6, pp. 17-26, 1995.

[5] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[6] IEEE Architecture Working Group, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE, 2000.

[7] Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.

[8] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, pp. 42-50, 1995.

[9] Land R., *Interviews on Software Systems Merge*, MRTC report, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[10] Land R. and Crnkovic I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", In *Information & Software Technology*, Accepted for publication, 2006.

[11] Land R. and Lakotic M., "A Tool for Exploring Software Systems Merge Alternatives", In *Proceedings of International ERCIM Workshop on Software Evolution* , 2006.

[12] Mens T., "A state-of-the-art survey on software merging", In *IEEE Transactions on Software Engineering*, volume 28, issue 5, pp. 449-462, 2002.

[13] Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.

[14] Parnas D. L., "Designing Software for Ease of Extension and Contraction", In *IEEE Transaction on Software Engineering*, volume SE-5, issue 2, pp. 128-138, 1979.

[15] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[16] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.

[17] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

Paper VI

Paper VI

This paper is a reprint of:

"A Tool for Exploring Software Systems Merge Alternatives", Rikard Land, Miroslav Lakotic, *Proceedings of International ERCIM Workshop on Software Evolution*, p 113-118, Lille, France, April, 2006

Abstract

The present paper presents a tool for exploring different ways of merging software systems, which may be one way of resolving the situation when an organization is in control of functionally overlapping systems. It uses dependency graphs of the existing systems and allows intuitive exploration and evaluation of several alternatives.

1. Introduction

It is well known that successful software systems has to evolve to stay successful, i.e. it is modified in various ways and released anew [11,15,16]. Some modification requests concern error removal; others are extensions or quality improvements. A current trend is to include more possibilities for integration and interoperability with other software systems. Typical means for achieving this is by supporting open or de facto standards [13] or (in the domain of enterprise information systems) through middleware [4]. This type of integration concerns information exchange between systems of mainly *complementary functionality*. There is however an important area of software systems integration that has so far been little researched, namely of systems that are developed in-house and *overlap functionally*. This may occur when systems, although initially addressing different problems, evolve and grow to include richer and richer functionality. More drastically, this also happens after company acquisitions and mergers, or other types of close collaborations between organizations. A new system combining the functionality of the existing systems would improve the situation from an economical and maintenance point of view, as well as from the point of view of users, marketing and customers.

1.1 Background Research

To investigate how organizations have addressed this challenge, which we have labeled *in-house integration*, we have previously performed a qualitative multiple case study [21] consisting of nine cases in six organizations.

At a high level, there seems to be four strategies that are analytically easy to understand [10]: *No Integration* (i.e. do nothing), *Start from Scratch* (i.e. initiate development of a replacing system, and plan for retiring the existing ones), *Choose One* (choose the existing system that is most satisfactory and

evolve it while planning for retiring the others), and – the focus of the present paper – *Merge* (take components from several of the existing systems, modify them to make them fit and reassemble them).

There may be several reasons for not attempting a *Merge*, for example if the existing systems are considered aged, or if users are dissatisfied and improvements would require major efforts. Reusing experience instead of implementations might then be the best choice. Nevertheless, *Merge* is a tempting possibility, because users and customers from the previous systems would feel at home with the new system, no or very little effort would be spent on new development (only on modifications), and the risk would be reduced in the sense that components are of known quality. It would also be possible to perform the *Merge* in an evolutionary manner by evolving the existing systems so that more and more parts are shared; this might be a necessity to sustain commitment and focus of the integration project. Among the nine cases of the case study, only in one case was the *Merge* clearly chosen as the overall strategy and has also made some progress, although there were elements of reuse between existing systems also in some of the other cases. Given this background research, we considered the *Merge* strategy to be the least researched and understood and the least performed in practice, as well as the most intellectually challenging.

1.2 Continuing with Merge

To explore the *Merge* strategy further, we returned to one of the cases and performed follow-up interviews focused on compatibility and the reasons for choosing one or the other component. The organizational context is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. The company conducts physics computer simulations as part of their core business, and both sites have developed their own 3D physics simulator software systems. Both systems are written in Fortran and consist of several hundreds of thousands lines of code, a large part of which are a number of physics models, each modeling a different kind of physics. The staff responsible for evolving these simulators is less than a handful on each site, and interviews with these people are our main source of information [9].

At both sites, there were problems with their model for a particular kind of physics, and both sites had plans to improve it significantly (independent of the merge). There was a strategic decision to integrate or merge the systems in the long term, the starting point being this specific physics module. This

study involved interviewing more people. It should be noted that although the interviewees met in a small group to discuss alternatives, they did not use our tool, since the tool has been created after, and partly influenced by, these events. The case is nevertheless used as an example throughout the present paper, to illustrate both the possibilities of the tool and motivate its usefulness in practice.

In an in-house integration project, there is typically a small group of architects who meet and outline various solutions [10]. This was true for the mentioned case as well as several others in the previous study. In this early phase, variants of the *Merge* strategy should be explored, elaborated, and evaluated. The rest of the paper describes how the tool is designed to be used in this context. The tool is not intended to automatically analyze or generate any parts of the real systems, only serve as a decision support tool used mainly during a few days' meeting. One important design goal has therefore been simplicity, and it can be seen as an electronic version of a whiteboard or pen-end-paper used during discussions, although with some advantages as we will show.

1.3 Related Work

Although the field of software evolution has been maturing since the seventies [11,16], there is no literature to be found on software in-house integration and merge. Software integration as published in literature can roughly be classified into: Component-Based Software Engineering [19,20], b) standard interfaces and open systems [13], and c) Enterprise Application Integration (EAI) [6,18]. These fields typically assume that components or systems are acquired from third parties and that modifying them is not an option, which is not true in the in-house situation. Also, these fields address components or systems *complementing* each other (with the goal of to reducing development costs and time) rather than systems that *overlap* functionally (with rationalization of maintenance as an important goal).

Although there are methods for merging source code [3,12], these approaches are unfeasible for merging large systems with complex requirements, functionality, quality, and stakeholder interests. The abstraction level must be higher.

We have chosen to implement a simple architectural view, the module view [5,7] (or development view [8]), which is used to describe development abstractions such as layers and modules and their relationships. Such dependency graphs, first defined by Parnas [14], are during ordinary

software evolution the natural tool to understand how modifications propagate throughout a system.

2. The Tool

The tool was developed by students as part of a project course. The foundation of the tool is a method for software merge. As this is ongoing work, this paper is structured according to the method but focuses on the tool. We also intend to publish the method separately, as it has been refined during the tool implementation – after which it is time to further improve the tool.

The method makes use of dependency graphs of the existing systems. There is a formal model at the core, with a loosely defined process on top based on heuristics and providing some useful higher-level operations. The tool conceptually makes the same distinction: there are the formally defined concepts and operations which cannot be violated, as well as higher-level operations and ways of visualizing the model, as suggested by the informal process. In this manner, the user is gently guided towards certain choices, but never forced. A fundamental idea with both the method and the tool is that they should support the exploratory way of working – not hinder it.

The actual tool is implemented as an Eclipse plug-in [1]. The model of the tool is based on the formal model mentioned above, and its design follows the same rules and constraints. The model was made using Eclipse Modeling Framework, and presented by Graphics Eclipse Framework combined using the Model-Controller-View architecture. This makes the tool adaptable and upgradeable.

2.1 Preparatory Phase

There are two preparatory activities:

Activity P-I: Describe Existing Systems. The user first needs to describe the existing systems as well as outline a desired future system. The current implementation supports two existing systems, but the underlying model is not limited to only two.

Activity P-II: Describe Desired Future Architecture. The suggestion of the final system is determined simply by choosing which modules are preferred in the outcome. Any system, A or B can then be experimented

upon, and the progress can be followed through a scenario tree. Figure 1 shows a snapshot of the tool with the two existing systems at the top and the future system at the bottom. It might be noted that the existing systems have – and must have – identical structures (this assumption is further discussed in section 2.3).

2.2 Exploratory Phase

The goal of the exploration is two system descriptions where some modules have been exchanged, so that the systems are evolved in parallel towards the desired future, merged system. The goal is not only to describe the future system (one graph would then be enough, and no tool support needed) but to arrive at next releases of the systems, in order to perform the merge gradually, as a sequence of parallel releases of the two existing systems until they are identical. This will involve many tradeoffs on the behalf of the architects (and other stakeholders) between e.g. efforts to be spent only on making things fit for the next release and more effort to include the more desired modules, which will delay next release of a system. The tool does not solve these tradeoffs but supports reasoning about them. There are four activities defined in the exploratory phase, with a rough ordering as follows, but also a number of iterations.

Activity E-I: Introduce Desired Changes. The starting point for exploration is to introduce some desired change. In the case, it was imperative to start by assuming a newly developed physics module (PX in the figures) to be shared by both systems. In other situations, the actual module to start with might not be given. In the tool, this is done by choosing the preferred module in the final system view, by clicking on the checkboxes. A new module can also be attached to the old system. This is done by clicking on the node in final system, and then clicking on the button "Create" in the *Actions View*. This will also require user input for the name of the new module and effort needed for its implementation (this could be zero for a pre-existing component such as a commercial or open source component, or a component to be reused in- house). After the module has been created, it can be used as any other module.The change to the system structure is made by clicking on the nodes and links in the input systems A and B. The modules the systems are using can be set up in the *Status View* for every node in any input system.

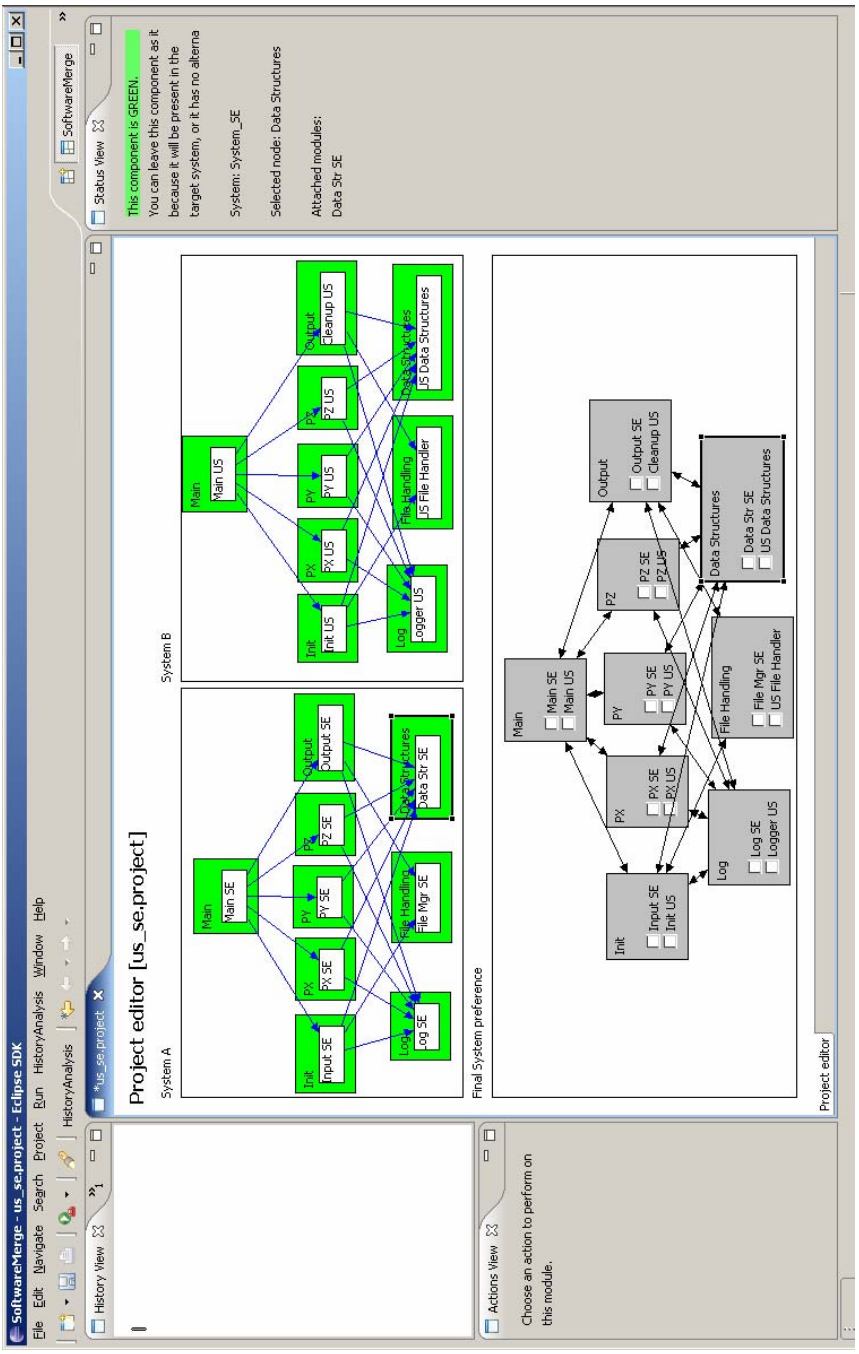


Figure 1: Initial systems state.

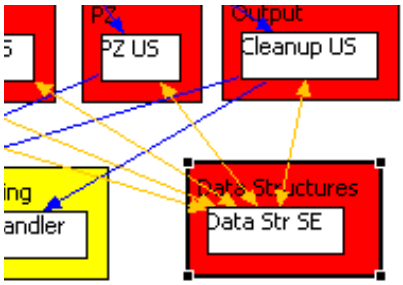


Figure 2: Example of highlighted inconsistencies.

Activity E-II: Resolve Inconsistencies. As changes are introduced, the tool will highlight *inconsistencies* between modules by painting the dependency arrows orange (see Figure 2). In the model, two module instances from the same system are consistent without further adaptation. Two modules from different systems are consistent only if some measure has been taken to ensure it, i.e., if either module have been adapted to work with the other. The actual adaptations made could in practice be of many kinds: some wrapping or bridging code as well as modifications of individual lines of code.

Another way to resolve an inconsistency is to describe adaptations to either of the inconsistent modules, in order to make them match. This is done by clicking on the incompatible link, and one of "Add …" buttons in the *Actions View*. This will require the user to enter an estimated effort for resolving this inconsistency (a number, in e.g. man-months), and a free text comment how to solve it, such as "we will modify each call to methods *x()* and *y()*, and must also introduce some new variables *z* and *w*, and do the current *v* algorithm in a different way" (on some level of detail found feasible). (As said, the tool does not do anything with the real systems automatically, but in this sense serves as a notebook during rapid explorations and discussions.) It can be noted that a module that will be newly developed would be built to fit. Nevertheless there is an additional complexity in building something to fit two systems simultaneously, which is captured by this mechanism.

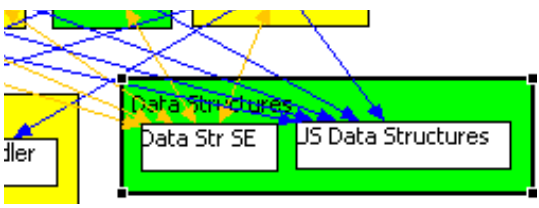


Figure 3: Two modules with same role.

There is also a third possibility to resolve an inconsistency: to let two modules for the same role live side by side, see Figure 3. Although allowing the same thing to be done in different ways is clearly a violation of the system's conceptual integrity, it could be allowed during a transition period (until the final, merged system is delivered) if the system's correct behavior can be asserted. For example, it might be allowed for some stateless fundamental libraries, but not when it is fundamentally assumed that there is only one single instance responsible for a certain functionality, e.g. for managing central resources, such as thread creation and allocation, access control to various hardware or software resources, security). The tool cannot know whether it would be feasible in the real system, this is up to the users to decide when and whether to use this possibility. The current version does not model the potential need for communication and synchronization of two modules doing same role.

Activity E-III: Branch Scenarios. As changes are made, the operations are added to a scenario tree in the *History View* (see Figure 4). At any time, it is possible to click any choice made earlier in the tree, and branch a new scenario from that point. The leaf of each branch represents one possible version of the system. When clicking on a node, the graphs are updated to reflect the particular decisions leading to that node. Any change to the systems (adaptations, exchanging modules, etc.) results in a new node being created; unless the currently selected node is a leaf node, this means a new branch is created. All data for adaptations entered are however shared between scenarios; this means that the second time a particular inconsistency is about to be resolved, the previous description and effort estimation will be used. As information is accumulated, the exploration will be more and more rapid.

Activity E-IV: Evaluate Scenarios. The exploration is a continuous iteration between changes being made (activities E-II and E-III) and evaluation of the systems. Apart from the information of the graphs themselves, the *Status View* presents some additional information, see Figure 5. The branching mechanism thus allow the architects to try various ways of resolving inconsistencies, undo some changes (but not loosing them) and explore several alternatives in a semi-parallel fashion, abandon the least promising branches and evaluate and refine others further. The total effort for an alternative can be accessed by clicking the "History Analysis" button, which is simply the sum of all individual adaptation efforts. It also becomes possible to reason about efforts related to modifications that actually lead towards the desired future system, efforts required only to make modules fit only for the next delivery (and later discarded).

The tool's advantage over using a whiteboard lies in the possibility to switch back and forth among (temporary) decisions made during the exploration (by means of the scenario tree), make some further changes (through simple point-and-click operations), and constantly evaluate the resulting systems (by viewing the graphs, the status view, and retrieve the total effort for the scenario).

Finally, although not implemented yet, one would extract the free texts associated with the scenario into a list of implementation activities.

2.3 Similar Structures?

The tool (and the model) assumes that the existing systems have identical structures, i.e. the same set of module roles (e.g. one module instance each for file handling, for physics X etc.) with the same dependencies between them. This may seem a rather strong assumption, but there are three motivations for this, based on our previous multiple case study [10]. First, our previous observations strongly suggest that similar structures are a prerequisite for merge to make sense in practice. Second, we also observed that it is not so unlikely that systems in the same domain, built during the same era, are indeed similar. And third, if the structures are not very similar, it is often possible to find a higher level of abstraction where the systems are similar.

With many structural differences, *Merge* is less likely to be practically and economically feasible, and some other high-level integration strategy should be chosen (i.e. *Start from Scratch* or *Choose One*). A common type of difference, that should not pose large difficulties in practice, is if there is a set of identical module roles and dependencies, and some additional modules that are only extensions to this common architecture. (For example, in the case we could imagine one of the systems to have a module modeling one more physics model PW than the other.) However, architects need in reality not be limited by the current version: a simple workaround solution is to introduce virtual module instances, i.e. modules that do not exist in the real system (which are of course not desired in the future system).

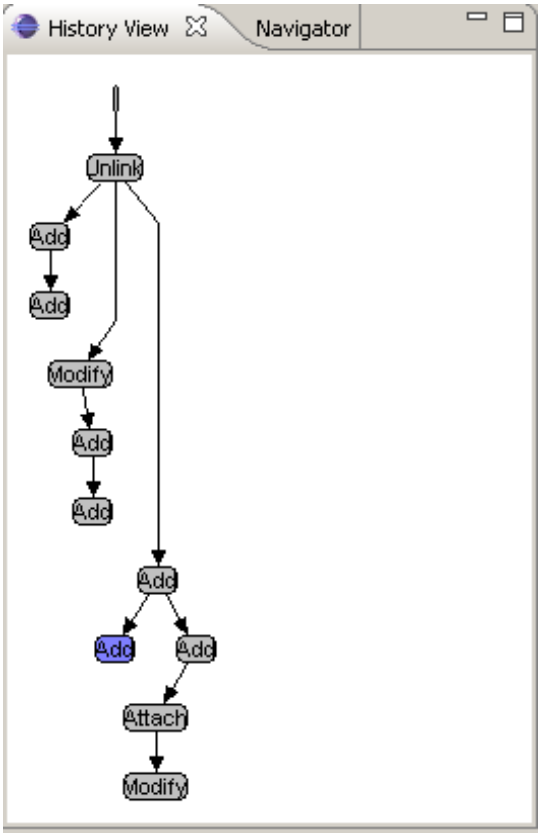


Figure 4: The History View.

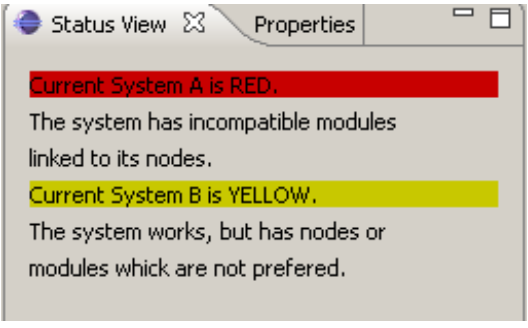


Figure 5: The Status View.

3. Future Research & Development

The tool is still in prototype stage and needs to be further developed. Neither the method nor the tool has been validated in a real industrial case (although their construction builds heavily on industrial experiences).

In reality there are numerous ways to make two components fit, for example as an adapter mimicking some existing interface (which requires little or no modifications of the existing code) or switches scattered through the source code (as runtime mechanisms or compile-time switches). Such choices must be considered by the architects: a high-performance application and/or a resource constrained runtime environment might not permit the extra overhead of runtime adapters, and many compile-time switches scattered throughout the code makes it difficult to understand. The method in its current version does not model these choices explicitly but has a very rough representation: the users can select which of the two inconsistent modules that should be adapted, and add a free text description and an effort estimation.

Another type of extension would be to include several structural views of the architecture, including some runtime view.

Yet another broad research direction is to extend the method and the tool to not focus so much on structure as the software architecture field usually does [2,17]. Structure is only one high-level measure of similarity between systems. Existing data models, and the technological frameworks chosen (in the sense “environment defining components”) are also important additional issues to evaluate [10], and needs to be included in any merge discussions in reality, and should be included in future extensions of the merge method and the tool.

4. Acknowledgements

We would like to thank the interviewees and their organization for sharing their experiences and allowing us to publish them. Thanks to Mathias Alexandersson, Sebastien Bourgeois, Marko Buražin, Mladen Čikara, Lei Liu, and Marko Pecić for implementing the tool. Also thanks to Laurens Blankers, Jan Carlson, Ivica Crnkovic, and Stig Larsson for previous and current research collaborations related to this paper.

5. References

[1] *Eclipse.org home*, URL: www.eclipse.org, 2006.

[2] Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.

[3] Berzins V., "Software merge: semantics of combining changes to programs", In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16, issue 6, pp. 1875-1903, 1994.

[4] Britton C. and Bye P., *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems* (2nd edition), ISBN 0321246942, Pearson Education, 2004.

[5] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[6] Cummins F. A., *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*, ISBN 0471400106, John Wiley & Sons, 2002.

[7] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[8] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, pp. 42-50, 1995.

[9] Land R., *Interviews on Software Systems Merge*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-196/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[10] Land R. and Crnkovic I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", In *Information & Software Technology*, accepted for publication, 2006.

[11] Lehman M. M. and Ramil J. F., "Software Evolution and Software Evolution Processes", In *Annals of Software Engineering*, volume 14, issue 1-4, pp. 275-309, 2002.

[12] Mens T., "A state-of-the-art survey on software merging", In *IEEE Transactions on Software Engineering*, volume 28, issue 5, pp. 449-462, 2002.

[13] Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.

[14] Parnas D. L., “Designing Software for Ease of Extension and Contraction”, In *IEEE Transaction on Software Engineering*, volume SE-5, issue 2, pp. 128-138, 1979.

[15] Parnas D. L., “Software Aging”, In *Proceedings of The 16th International Conference on Software Engineering*, pp. 279-287, IEEE Press, 1994.

[16] Perry D. E., “Laws and principles of evolution”, In *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 70-70, IEEE, 2002.

[17] Perry D. E. and Wolf A. L., “Foundations for the study of software architecture”, In *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, pp. 40-52, 1992.

[18] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[19] Szyperski C., *Component Software - Beyond Object-Oriented Programming* (2nd edition), ISBN 0-201-74572-0, Addison-Wesley, 2002.

[20] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.

[21] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

Appendices

Appendix A: Questionnaire Form and Data for Phase One

This appendix reprints the questionnaire forms and data for phase one. The questionnaire for all participants is reprinted first, followed by additional questions for managers only (on page 278). All references to the company have been removed from the answers (replaced with an indication in brackets, like "<*system 1 name*>".

This questionnaire aims at understanding why the project executed during October through December (henceforth "the project") succeeded in making at a decision while the first sets of meetings held earlier in 2002 ("the meetings") failed.

You should answer the questions by marking an "X" in the first column for the alternative(s) you agree with. There is an empty row for you to write your own alternatives in free text. All comments and clarifications are very welcome!

In any publication of these results, including internally at <*company name*>, you are guaranteed anonymity.

Thank you for your cooperation!

Rikard Land

1.

Which of the following meetings did you participate in? (Please mark zero or more meetings with "X".)

						Meeting(s) on: ***dates***
			x			Project phase 1 (users)
			x	x	X	Project phase 1 (developers)
	X	X		x	X	Project phase 2
X		X	X			Project phase 3

Comment:

- I listened to meeting summaries during phases 1 and 2 but was not a direct participant (other than buying some lunches & dinners!)

2.

The project made an explicit separation of the activities so that users, developers, and managers met separately. Compare this with the previous meetings, where these "roles" met together. Considering only the separation of people (and not the time spent) which of the following statements do you agree with? (Please mark zero or more statements with "X".)

X			X	x	X	This made the users' evaluation more efficient and focused (phase 1).
X			X	x	X	This made the developers' evaluation more efficient and focused (phase 1)
X		X		x		This made the developers' design and analysis discussions more efficient and focused (phase 2)
X		X	X		X	This made the managers' discussions more efficient and focused (phase 3)

Comment:

- Please note that the groups were not entirely "separate" – the users & developers met during Phase 1 and heard first hand the feedback and recommendations.

3.

Which of the following statements would you agree with? (Please mark only one statement with "X".)

			X			In the first sets of meetings, the responsibilities were unclear, which was one reason that we could not agree
	X			x		In the first sets of meetings, the responsibilities were unclear, but this was not significant enough to affect the outcome
						In the first sets of meetings, the responsibilities were clear
		X				I did not participate

Comment:

- The overall project scope and commitment were not clear. Reasonable technical judgement can not be made.
- The second set of meetings didn't differ from the first set in the sense that we agreed on everything. In the second set we left some questions

unanswered (Tcl/Java GUI etc.). What was good in the second meetings was that we spent more time on the project plan.

- I thought we had agreement to use the <*system 3 name*> architecture and Java for new user interfaces during the June meetings but that changed when we re-grouped for the September meetings when some participants wanted Tcl for all user interface development (and perhaps no <*system 3 name*> architecture since it might not work with Tcl). The mission and responsibilities were clear – we just could not reach agreement on an approach.

4.

In your opinion, how important is it that time had passed (6 months) from the first attempts? (Please mark only one statement with "X".)

	X			x		Another meeting similar to the first ones had succeeded in making a decision, if held in October/November
		X	X		X	Another meeting similar to the first ones had not succeeded in making a decision

Comment:

- Some decisions were made, but the "project" is still in the air.
- 6 months had not passed from the first attempts – the first set of meetings took place in April, June and beginning of September. The user & developer evaluations were initiated at the end of September, so there was only about 3 weeks of delay, which had no impact on making the decision in my opinion.

5.

In your opinion, how important was the stronger requirement from <*senior manager*> to make a decision before the end of the year? (Please mark only one statement with "X".)

						A short meeting with the managers had been enough to make a decision
X	X	X	X	x	X	The project was essential, although it was costly

Comment:

- The sooner the decision is made the better for the company.
- Although the decision to create common data (database) models is not enough. (my personal opinion).

6.

The decision was quite costly, in terms of time spent by the people involved. In your opinion, could any of the following (cheaper) decision processes have succeeded in making a decision and gained support enough for it to actually make it happen? (Please mark zero or more statements with "X".) Also describe the level of user involvement required.

						One single architect could have been assigned the task of developing alternatives and decide which to use (no separate decision needed)
						One single architect could have made the design, and one single manager could have decided
						One single architect could have made the design, and several managers could have agreed on a decision
	X					None of the above, because: *We needed the knowledge of the designers/users involved. There are no designers/users that has knowledge of all existing systems.*
				x		None of the above, because: *A smaller set of people would not have been aware of all of the issues. Any decision they would have made would still have been second guessed.*
X						Other constellations: *Given that automation is a key contributor to our future I think it was important to have all of the impacted organizations involved so that there is appropriate "buy-in" to the recommendation.*

			x			None of the above, because: *there was not one single architect that knew enough about all of the available systems*
		X				None of the above, because: *the real issue was not technical but cultural and you needed time to listen to one another and "buy-in" to a compromise solution*
						Other constellations: <free text>

User(s) should be involved in the following way as a minimum:

I think the users were appropriately involved (during Phase 1).
Defining the requirements, testing the system
See above.

Comments:

- "One single architect" solution might be an feasible solution. Integration and evolution are the key, not re-writing.
- The integration framework, and integration phases can be decided in some of the early meetings.
- A manager from the high level (<*senior manager*> level) should make the "go" decision 6 month ago.

7.

Which of the following statements do you agree with? (Please mark only one statement with "X".)

	X					The architectural design solution decided upon was the best
		X	x	x		There was at least one architectural design proposal (not necessarily the one recommended in phase 2) that was better than the one decided upon. (If you answered "X" and you were part of phase 3, please indicate the reason you agreed in the decision.)

X						*The decision was not based on the "best" architectural design solution but on a solution that would minimize risk, leverage already existing applications at the different sites, and provide an affordable path to an acceptable solution.*

Comments:

- Don't really understand the question. See comments at the bottom.
- The recommendation of Phase 2 was the best architectural design but not practical to implement given the available resources
- Being technically the best does not mean that it was the best overall route. There are many other equally important factors such as organizational readiness to execute the strategy selected.
- There was no decision of language for the GUI. This decision impacts how much <*system 1 name*> code can be reused.

8.

Considering the people involved, which of the following was the most important difference between the first sets of meetings and the last project? (Please mark only one statement with "X".)

						The number of people was more important than the combination of people
X	X	X	X	x	X	The combination of people was more important than the number of people

Comments:

- The right group of people is always the starting point.
- In the second set of meetings, the users spend 2 week, but I really don't think they found out much new.
- Users were not involved in the first set of meetings and their input was important on accepting multiple look/feel in the User Interfaces.

9.

How devoted to the software integration were you at the time of the project? (Please mark zero or more statement with "X".)

					X	I did not prepare myself enough in advance of the project phases I participated in
	X				X	I had wanted to spend more time on the project
					X	During the project, I had other more important work to do
						I hope I will not be more involved
			X			*I gave it my best.*
					X	*I hope I had more time to prepare.*

Comment:

- I had almost zero time to prepare for the meeting. It is true for all the <*larger project name*> meetings I attended. There was a lot of other work going on at the time the meetings were held. I did't find a shop order to work on.
- There should be more resource dedication to this project.
- I'm not sure you realise how difficult it is to build consensus among three different groups of people that have different experiences, backgrounds, and objectives. Not every decision is based purely on the technical aspects of the problem. The difference between a scientist and an engineer is economics. Based on the tone of your questions you are trying to be a scientist. You need to consider the economics of the situation to develop a plan that can be implemented. Then you become an engineer. I hope this helps you to understand what we have done.
- I devoted whatever time was needed for all the project sessions that I participated in and made them a priority

10.

Please describe what you think the outcome of the integration effort will be (for example, "according to the project plan discussed", or "too late and too expensive", or "never").

According to the project plan discussed
As I said above, a common data model in itself is not enough. I hope and

believe that more will follow (common server architecture etc.).
Given history as our guide, some small accomplishment will be made but the true integrated vision will not be achieved due to the lack of strategic funding and priority. Another key factor is the ever-changing landscape of executive leadership with accompanying changes in direction and emphasis.
I hope it will be according to the project plan discussed. It will probably be later than projected however.
It is a big project. It is necessary. It is expensive. It requires company management support to make it happen. I still doubt the effort vs. gain of combining <*domains*> in the same tool.
too late and too expensive

11.

Please note any other comments or experiences that you would like to share.

From the software engineering point of view, <*system 3 name*> architect is a better design any other systems we investigated. As a matter of fact, the <*system 3 name*> team will make a Birds-of-a-Feather presentation at <*conference name*> about the integration work we did in <*system 3 name*>. The key of the project is integration, not re-write. <*system 3 name*> should not replace the existing <*system 1 name*> for all. The whole system (the New System, or expanded <*system 3 name*>) should be componentized, and different legacy applications should be able to live within.
I think that the reason that we obtained consensus on a decision the second time was because management insisted that we do so. By way of clarification, the process used earlier in 2002 included 3 sets of meetings which involved both manager & developers (no users) in April, June & September (each lasted ~4 days).

Questions for managers only

This questionnaire aims at understanding why the project executed during October through December (henceforth "the project") succeeded in making at a decision while the first sets of meetings held earlier in 2002 ("the meetings") failed.

You should answer the questions by marking an "X" in the first column for the alternative(s) you agree with. There is an empty row for you to write your own alternatives in free text. All comments and clarifications are very welcome!

In any publication of these results, including internally at <*company name*>, you are guaranteed anonymity.

Thank you for your cooperation!

Rikard Land

1.

Why did <u>you</u> choose the non-recommended solution? (Please mark zero or more statement with "X".)

	x	It means a lower degree of commitment
	x	I would have preferred the recommended solution, but had to compromise
		The project plan the developers produced for the recommended solution was not realistic
		The developers seemed to recommend the solution they did because it was more "elegant" but no more useful
X	x	It would mean a higher risk to rewrite code in a new language (as in the solution recommended by phase 2)
X		It would mean a higher cost to rewrite code in a new language (as in the solution recommended by phase 2)
	x	

Comments:

2.

Why do you think the other managers chose the non-recommended solution? (Please mark zero or more statement with "X".)

	x	It means a lower degree of commitment
		Some of them would have preferred the recommended solution, but they had to compromise
		They judged the project plan the developers produced for the recommended solution to be unrealistic
		They thought the developers recommended the solution they did because it was more "elegant" but no more useful
X		They thought it would mean a higher risk to rewrite code in a new language
X		They thought it would mean a higher cost to rewrite code in a new language
	X	*The level of commitment over time required to complete the plan was not likely to be found within the organization*

Comments:

3.

Managers only. What were the developers' reasons, in your opinion, to recommend the use of one single language (Java) on the server side, and one single language (Java or Tcl) on the client side, instead of a multi-language solution? (Please mark zero or more statement with "X".)

X	X	To give the user interface a homogeneous look-and-feel
X	X	Because the same functionality would otherwise have to be duplicated in several languages
X	X	Because that would simplify integration of components
X		Because some of the languages and technologies used today are very old-fashioned and this will give rise to problems in the future
		Because some of the languages and technologies used today are very old-fashioned and the solution would be less "elegant"
X	X	To simplify maintenance

X		To simplify cooperation between Sweden and the US
		To easier attract and keep staff
		To enable the application to be run via the Internet
		Because Java is well suited for number-crunching
	X	Because Java and Tcl are well suited for writing user interfaces
		The total number of lines of code would be considerably less

Comments:

4.

The solution recommended by phase 2 was said to use one single architecture, and reflect one set of concepts and design decisions. Which of the following statements do you agree with? (Please mark zero or more statements with "X".)

		This remark is irrelevant for a decision
		The decision must be based on quantified cost savings, not vague remarks like this
X		This was an important remark which I considered seriously
	x	*I sense from the choices above that you feel we made the wrong decision. We considered this remark but in the end decisions must be made based on financial terms and the likelihood of completing the project. It was very likely that if we recommended the very expensive solution suggested that we would fail to get any financial support and the project never even get started. Given that I think we made the right choice.*

Comments:

- This is an important concept which would be much more readily achievable if we were starting from "scratch" rather than having to evolve from existing applications. However, I didn't think the organization would be willing to invest the funding necessary to realize this goal in a short time period.

5.

Why did you lower the estimated costs for the chosen solution, compared to the developers' estimations? (Please mark only one statement with "X".)

		I do not know that we did
	X	Some costs were associated with activities we can do without
		It just has to be cheaper

Comments:

- I believe the costs for rewriting the applications will be much higher than the developers' estimates.

6.

Why, in your opinion, was not the project design that succeeded used earlier during 2002? (Please mark zero or more statement with "X".)

		No one thought of it
		It was too expensive
		A meeting such as the ones that were held was believed to succeed
		It was considered better to mix users, developers, and managers in one meeting than separating them
	x	*I don't understand what you are asking*

Comments:

- Because we thought we could get group consensus quickly by getting all of the right people together. In retrospect this was a fantasy.
- I don't understand this question – which project design succeeded?

7.

From a psychological point of view, it might have been unfortunate to decide to use a solution that the developers (who will have to implement it) explicitly did not recommend. Which of the following statements do you agree with? (Please mark zero or more statements with "X".)

		I do not understand what why it should be unfortunate
		The developers' opinions are not relevant – a company is not a democracy
X		Yes, this may be a source of conflicts, but the chosen solution was so much better so it is worth this risk
		Yes, this may be a source of conflicts, but it was the only way we could agree
		We did not think of this during the meeting in phase 3
	X	*The developers would be even more disappointed if the project never gets started. See my comments above in #4*

Comments:

- All of us must get behind and support the decisions of our management, regardless of whether we agree with them. I would encourage the developers to continue to raise concerns but also be open to alternative solutions that are less than ideal.

8.

Have you changed the plans for your department to be able to perform the integration (e.g. allocated staff and planned journeys for your employees)? (Please mark only one statement with "X".)

X	x	Yes
		No
		No, because it has not been needed – the previous plans already incorporated this possibility

Comments:

9.

Do you believe the other managers have changed their plans? (Please mark only one statement with "X". If more than one answer is selected, please clarify.)

X	X	Yes
		No
		No, because it has not been needed – the previous plans already incorporated this possibility

Comments:

Appendix B: Interview Questions for Phase Three

This appendix reprints the interview questions used in phase three.

The present research is intended to investigate *integration of software systems*. Of interest are integration of systems that have a significant complexity, both in terms of functionality, size and internal structure, and which have been released and used in practice. Our research question is: what are feasible *processes* (such as when and how were different people involved in the process) and *technical solutions* (for example, when is reuse possible, and when is rewrite needed) to accomplish a successful integration?

The working hypothesis is that both processes and technical solutions will differ depending on many factors: the fundamental reasons to integrate, as well as the domain of the software, the organizational context, and if there are certain very strict requirements (as for safety-critical software). We aim to identify such factors and their importance. In particular, we are interested in the role of *software architecture* (that is, the systems' overall structure) during integration.

The questions below will be used rather informally during a discussion/interview, and are to be used as a guide. Preferably, the respondent has considered the questions in advance. It is not necessary that all terms be understood. There may also be other highly relevant topics to discuss.

1. Describe the technical history of the systems that were integrated: e.g. age, number of versions, size (lines of code or other measure), how was functionality extended, what technology changes were made? What problems were experienced as the system grew?

2. Describe the organizational history of the systems. E.g. were they developed by the same organization, by different departments within the same organization, by different companies? Did ownership change?

3. What were the main reasons to integrate? E.g. to increase functionality, to gain business advantages, to decrease maintenance costs? What made you realize that integration was desirable/ needed?

4. At the time of integration, to what extent was source code the systems available, for use, for modifications, etc.? Who owned the source code? What parts were e.g. developed in-house, developed by contractor, open source, commercial software (complete systems or

smaller components)?

5. Which were the stakeholders* of the previous systems and of the new system? What were their main interests of the systems? Please describe any conflicts.

6. Describe the decision process leading to the choice of how integration? Was it done systematically? Were alternatives evaluated or was there an obvious way of doing it? Who made the decision? Which underlying information for making the decision was made (for example, were some analysis of several possible alternatives made)? Which factors were the most important for the decision (organizational, market, expected time of integration, expected cost of integration, development process, systems structures (architectures), development tools, etc.)?

7. Describe the technical solutions of the integration. For example, were binaries or source code wrapped? How much source code was modified? Were interfaces (internal and/or external) modified? Were any patterns or infrastructures (proprietary, new or inherited, or commercial) used? What was the size of the resulting system?

8. Why were these technical solutions (previous question) chosen? Examples could be to decrease complexity, decrease source code size, to enable certain new functionality.

9. Did the integration proceed as expected? If it was it more complicated than expected, how did it affect the project/product? For example, was the project late or cost more than anticipated, or was the product of less quality than expected? What were the reasons? Were there difficulties in understanding the existing or the resulting system, problems with techniques, problems in communication with people, organizational issues, different interests, etc.?

---

* Stakeholders = people with different roles and interests in the system, e.g. customers, users, developers, architects, testers, maintainers, line managers, project managers, sales persons, etc.

10. Did the resulting integrated system fulfill the expectations? Or was it better than expected, or did not meet the expectations? Describe the extent to which the technical solutions contributed to this. Also describe how the process and people involved contributed – were the right people involved at the right time, etc.?

11. What is the most important factor for a successful integration according your experiences? What is the most common pitfall?

12. Have you changed the way you work as a result of the integration efforts? For example, by consciously defining a product family (product line), or some components that are reused in many products?

Appendix C: Interview Questions for Phase Four

This appendix reprints the interview questions used in phase four.

Structure

Describe the structure of the system. What components are there? What are their roles? How are they connected; how is data and control transferred? (Is there any documentation of this?)

Framework

How are components defined? Do you utilize e.g. any language or operating system constructs? To what extent can modularity be enforced? To what extent do you rely on conventions (e.g. different files/directories with standardized names)?

Conceptual Integrity

For each of the following concepts *X*:

a) Error handling
b) Physics PX
c) Data structures
d) More?

Describe:

1. The *X* component

- Today and in the future:
  - Functionality
  - Interface
- How did you define the future component, in terms of the existing POLCA/ANC *X* component? To what extent did you try to achieve some similarity with today, and to what extent did you try to create something as good as possible?
- What will it take to move from today's component to the future *X* component?
  - How difficult will it be to modify the system to always use the new *X* component?
  - Did you assess this explicitly?

2. Any rules associated with *X* (which the whole system must follow):

- Today and in the future:
  - What does *X* required from the rest of the system?
  - What does *X* prohibit?
  - What would happen if these rules are not followed? Would the run-time behaviour be unpredictable/error prone, and/or would the system becomes more difficult to maintain?

- Are these rules documented? Are they known? Do you enforce these rules in any way?

- How did you define the future rules, in terms of the existing POLCA/ANC *X* rules? To what extent did you try to achieve some similarity with today, and to what extent did you try to create something as good as possible?
- In terms of rules, what will it take to move from today ’s system to the future *X*?
  - How difficult will it be to modify the system to always use the new *X*?
  - Did you assess this explicitly?

Appendix D: Questionnaire Form for Phase Five

This appendix reprints the questionnaire. For questions 76-101, the "importance" column is assigned the even question ID and "attention" the odd number; e.g. for the statement "A small group…", "importance" has ID 76 and "attention" 77.

This questionnaire is aimed at studying various aspects of the integration, including how decisions are made, the technical nature of the systems and the integration, and certain practices. Please answer to the best of your knowledge. You do not need to provide any free-text comments, but you are free to communicate anything with us – clarifications, comments on the formulation of questions, or similar.
There are four main sections, labeled A-D, with a total of 101 questions. The questionnaire is expected to take ca 20 minutes to fill. All answers will be treated anonymously and confidentially.
As this questionnaire is distributed to projects in various stages of the integration, we want to clarify that "existing systems" refer to the original systems, that have been or are to be integrated. "Future system" is the system resulting from the integration (it may already exist as well, if the integration is completed.

First we ask you to fill some background information.		
1	Project Name	
2	My experience in software development activities	Years
3	My experience with any of the existing systems	Years
Please mark your role(s) in the current project with "X".		
4	(Technical) architect	[ ]
5	Designer	[ ]
6	Implementer	[ ]
7	Tester	[ ]
8	Project leader	[ ]
9	Line manager	[ ]
10	Product responsible/owner	[ ]
11	Other	[ ]

Comments	

Section A.

You will now be asked some questions concerning management, how decision was reached, and how the existing systems will eventually be integrated.		
The following questions concern what, in your opinion, management's vision is of your project, i.e. the high-level decision about how to integrate.		
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".		
12	The existing systems will continue to be maintained, evolved and deployed completely separately.	1 2 3 4 5 NA
13	One of the existing systems is (or will be) evolved into a common system.	1 2 3 4 5 NA
14	One or more systems has been (or will be) retired.	1 2 3 4 5 NA
15	All existing systems is (or will be) retired.	1 2 3 4 5 NA
16	A new generation of this type of systems is (or will be) developed from scratch.	1 2 3 4 5 NA
17	Parts/components/modules of the future system are (or will be) reused from more than one of the existing systems.	1 2 3 4 5 NA
18	Reused parts/components/modules required (or will require) only minor modifications	1 2 3 4 5 NA
19	A significant amount of the existing systems are (or will be) reused in the future system	1 2 3 4 5 NA
20	The functionality of the existing systems are equal.	1 2 3 4 5 NA
21	The quality of the existing systems are equal.	1 2 3 4 5 NA
22	At least some software parts/components/modules is (or will be) completely new	1 2 3 4 5 NA
23	Source code is (or will be) reused from one or more of the existing systems.	1 2 3 4 5 NA

The following questions concern how, in your opinion, this vision was reached.
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".

The high-level decision about how to integrate…		
24	…was based on technical considerations	1 2 3 4 5 NA
25	...was based on considerations on time schedule	1 2 3 4 5 NA
26	…was based on considerations for existing users	1 2 3 4 5 NA
27	…was based on considerations concerning the parallel maintenance and evolution of existing systems	1 2 3 4 5 NA
28	…was based on available staff and skills	1 2 3 4 5 NA
29	...was based on politics	1 2 3 4 5 NA
30	...was made by technicians	1 2 3 4 5 NA
31	...was made by management	1 2 3 4 5 NA

Now some questions about your personal opinion about what you think will happen (or have happened) in the project, i.e. how the systems will actually be integrated. This could be identical or different from management's vision/decision.		
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".		
32	The existing systems will continue to be maintained, evolved and deployed completely separately.	1 2 3 4 5 NA
33	One of the existing systems is (or will be) evolved into a common system.	1 2 3 4 5 NA
34	One or more systems has been (or will be) retired.	1 2 3 4 5 NA
35	All existing systems is (or will be) retired.	1 2 3 4 5 NA
36	A new generation of this type of systems is (or will be) developed from scratch.	1 2 3 4 5 NA
37	Parts/components/modules of the future system are (or will be) reused from more than one of the existing systems.	1 2 3 4 5 NA
38	Reused parts/components/modules required (or will require) only minor modifications	1 2 3 4 5 NA
39	A significant amount of the existing systems are (or will be) reused in the future system	1 2 3 4 5 NA
40	The functionality of the existing systems are equal.	1 2 3 4 5 NA

41	The quality of the existing systems are equal.	1 2 3 4 5 NA
42	At least some software parts/components/modules is (or will be) completely new	1 2 3 4 5 NA
43	Source code is (or will be) reused from one or more of the existing systems.	1 2 3 4 5 NA

Comments	

Section B. Reuse and retirement

Now follows a number of questions concerning retirement of the existing system and backward compatibility of the final system. (All questions about retiring systems refer to the implementations, not how the systems are named or marketed.)		
The following questions concern what, in your opinion, management's vision is of your project.		
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".		
44	None of the existing systems will be retired.	1 2 3 4 5 NA
45	One or more existing system will be retired.	1 2 3 4 5 NA
46	There will be a replacement system that covers all the lost functionality of retired system(s).	1 2 3 4 5 NA
This decision was based on the opinions of...		
47	…customers	1 2 3 4 5 NA
48	…users	1 2 3 4 5 NA
49	…developers	1 2 3 4 5 NA
50	…marketing people	1 2 3 4 5 NA
51	…management	1 2 3 4 5 NA

Now some questions about your personal opinion about what you think will happen (or have happened) in the project, i.e. how the systems will actually be integrated. This could be identical or different from management's vision/decision.		
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".		
52	None of the existing systems will be retired.	1 2 3 4 5 NA
53	One or more existing system will be retired.	1 2 3 4 5 NA
54	There will be a replacement system that covers all the lost functionality of retired system(s).	1 2 3 4 5 NA

The following questions concern what, in the project, are (or were) important aspects of backward compatibility.
Please grade the statements below using the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".

The future system needs to...		
55	...support the way users currently work.	1 2 3 4 5 NA
56	...be backwards compatible with existing data.	1 2 3 4 5 NA
57	...be backwards compatible with existing surrounding tools.	1 2 3 4 5 NA
58	...be backwards compatible with installations of the existing systems.	1 2 3 4 5 NA

Comments	

Section C. The existing systems

Now follows a number of questions concerning the existing systems.		
Please grade the statements below according to how well, in your opinion, they describe the existing systems in your project.		
Use the scale 1-5, where 1 means "I do not agree at all" and 5 "I agree completely". NA means "cannot answer".		
59	The software of the existing systems have the same internal structure (architecture).	1 2 3 4 5 NA
60	The parts/components/modules exchange data in the same ways in the existing systems.	1 2 3 4 5 NA
61	The existing systems interacts with the users in the same way.	1 2 3 4 5 NA
62	The existing systems have similar look-and-feel of the user interface.	1 2 3 4 5 NA
63	The existing systems contain software parts/components/modules with similar functionality.	1 2 3 4 5 NA
64	The hardware topology (networks, nodes) of the systems is similar.	1 2 3 4 5 NA
65	The design of the existing systems is based on the data model.	1 2 3 4 5 NA

66	The data models in the existing systems are similar.	1 2 3 4 5 NA
67	The implementations of data handling in the existing systems are similar.	1 2 3 4 5 NA
68	The existing systems are written in the same programming language.	1 2 3 4 5 NA
69	Communication between components/modules/parts in the existing systems is performed through certain interfaces.	1 2 3 4 5 NA
70	The existing systems use some technology to clearly encapsulate software components/modules/parts.	1 2 3 4 5 NA
71	The existing software use the same or similar technologies.	1 2 3 4 5 NA
72	The existing systems implement some domain standards.	1 2 3 4 5 NA
73	The existing systems implement the same domain standards.	1 2 3 4 5 NA
74	The existing systems were initially built in the same time period (e.g. decade).	1 2 3 4 5 NA
75	The existing systems have evolved from the same system many years ago.	1 2 3 4 5 NA

Comments	

Section D. Practices

Now follows a number of questions concerning specific practices.			
For each statement below, please indicate the following: how important it was (or would have been) for your project's success, and how much attention it was given in your project.			
Please use the scale 1-5. For "importance" 1 means "not important at all" and 5 means "essential for success. For "attention", 1 means "no attention was given" and 5 "very much attention was given". The same grade on both "importance" and "attention" means that with respect to importance, enough attention was given but not too much. NA means "cannot answer".			
76	A small group of experts must be assigned early to evaluate the existing systems and describe alternative high-level strategies for the integration.	**Importance** 1 2 3 4 5 NA	**Attention** 1 2 3 4 5 NA
78	Experience of the existing systems from many points of view must be collected.	1 2 3 4 5 NA	1 2 3 4 5 NA

80	The future system should be described in terms of the existing systems.	1 2 3 4 5 NA	1 2 3 4 5 NA
82	The future system must contain more features than the existing systems	1 2 3 4 5 NA	1 2 3 4 5 NA
84	Decisions should wait until there is enough basis for making a decision	1 2 3 4 5 NA	1 2 3 4 5 NA
86	It is more important that decisions are made in a timely manner, even if there is not enough basis for making a decision	1 2 3 4 5 NA	1 2 3 4 5 NA
88	A strong project management is needed	1 2 3 4 5 NA	1 2 3 4 5 NA
90	All stakeholders must be committed to the integration	1 2 3 4 5 NA	1 2 3 4 5 NA
92	Management needs to show its commitment by allocating enough resources	1 2 3 4 5 NA	1 2 3 4 5 NA
94	The “grassroots” (i.e. the people who will actually do the hard and basic work) must be cooperative, both with management and each other	1 2 3 4 5 NA	1 2 3 4 5 NA
96	Formal agreements between sites must be made and honored (strictly obeyed)	1 2 3 4 5 NA	1 2 3 4 5 NA
98	A common development environment is needed	1 2 3 4 5 NA	1 2 3 4 5 NA
100	There is a conflict between the integration efforts and other development efforts	1 2 3 4 5 NA	1 2 3 4 5 NA

Comments	

Thank you for your participation! Your answers will be treated anonymously and confidentially.

Appendix E: Questionnaire Data for Phase Five

In this appendix, the complete questionnaire data is listed. (Respondent IDs are assigned by the order in which they were received.)

Respondent ID / Question ID	*3*	*1*	*9*	*2*	*4*	*5*	*6*	*7*	*8*
1	A	B	C	E1	E2	F2	F4	G	G
2	23	25	20	12	19	5	20	10	23
3	5	5	10	4	19	0		4	15
4	x							x	
5		x		x		x		x	
6		x		x	x	x		x	
7		x			x			x	
8	x	x	x	x	x			x	
9		x	x				x		
10		x	x						x
11					x				
12	2	2	1	1	3	2	3	4	5
13	5	4	5	1	2	5	4	4	3
14	3	5	5	5	4	5	4	2	5
15	1	1	5	2	1	1	1	5	1
16	2	4	3	4	1	3	1	5	5
17	1	2	4	4	4	4	5	2	4
18	1	3	3	1	2	1	3	2	4
19	1	1	4	3	4	4	4	2	4
20	3	3	4	1	2	3	3	3	4
21	3	1	5	1	2	2	3	3	1
22	5	5	5	4	4	5	5	5	5
23	1	1	3	1	5	4	5	2	1
24	3	4	4	4	4	2	3	3	5
25	1	1	2	1	2	4	1	4	1

Respondent ID / Question ID	*3*	*1*	*9*	*2*	*4*	*5*	*6*	*7*	*8*
26	1	4	4	1	5	3	2	4	3
27	3	4	4	1	5		4	2	2
28	3	3	4	2	2	3	3	3	4
29	3	2	3	1	4	5	3	2	4
30	4	1	4	4	4	2	2	3	4
31	4	4	4	4	5	5	4	3	5
32	2	1	1	1	3	3	4	4	5
33	4	3	5	1	2		4	4	1
34	5	5	3	5	5	2	2	4	5
35	2	2	3	2	1	2		2	1
36	1	4	3	5	4	3	1	5	5
37	1	2	4	1	4	3	4	2	5
38	1	3	2	1	2	1	3	2	5
39	1	1	4	2	4	2	3	1	5
40	3	3	4	1	2	3	3	3	4
41	3	1	5	1	2	2	3	3	1
42	5	5	5	5	4	4	5	5	5
43	1	1	2	1	4	2	5	1	1
44	1	1	1	1	1		2	2	1
45	5	5	5	5	5	4	2	5	5
46	4	4	4	4	2	5	2	5	5
47	3	2	5	4	5	4	2	3	4
48	3	4	5	4	5	4	2	1	2
49	4	4	2	4	4	2	2	4	5
50	4	1	2	NA	1	3	1	5	4
51	4	4	5	4	1	4	2	5	4
52	1	1	1	1	1	4	2	2	1
53	5	5	5	4	5	2	3	5	5
54	4	4	5	4	1	4	1	5	5
55	4	3	5	3	4	3	4	2	5
56	1	5	4	1	3	2	4	2	3
57	4	4	4	1	4	4	3	2	4
58	3	4	3	1	2	2	4	2	2
59	1	2	NA	1	2	4	1	2	1
60	3	1	2	1	1	4	1	2	1
61	4	2	4	2	1	3	1	3	4
62	5	2	4	1	2	3	1	3	1

Respondent ID / Question ID	*3*	*1*	*9*	*2*	*4*	*5*	*6*	*7*	*8*
63	4	4	4	4	5	4	3	3	4
64	2	3	4	1	4	5	2	4	2
65	4	3	NA	NA	4	4	2	NA	1
66	4	3	NA	NA	4	4	1	2	1
67	2	3	2	4	4		1	2	2
68	1	2	4	1	5	3	1	3	3
69	5	5	NA	3	3		1	2	4
70	5	3	NA	3	3		1	2	1
71	3	2	2	2	5	4	1	4	2
72	4	3	NA	NA			3	2	3
73	3	3	NA	NA		NA	1	2	1
74	3	3	2	3	5	NA	1	4	1
75	3	2	5	3	3	NA	5	5	1
76	4	5	5	5		5	5	4	4
77	3	5	5	5		3	2	NA	3
78	5	4	5	5	5	4	4	4	5
79	3	3	4	5	5	4	2	NA	3
80	4	3	4	1			3	2	2
81	3	3	4	1			NA	2	2
82	3	2	3	1	2	3	5	4	3
83	3	4	3	1	2	3	3	NA	4
84	3	4	5	5	4	5	3	5	2
85	3	4	4	5	3	2	2	2	1
86	4	3	4	5	4		4	2	4
87	2	3	4	5	3		3	NA	1
88	5	5	5	4		5	5	4	5
89	4	5	4	4		2	3	3	2
90	5	4	4	5		4	5	4	
91	2	4	5	3		NA	2	NA	
92	5	5	5	4		5	5	5	5
93	2	4	4	4		2	2	3	2
94	5	4	5	5	4	4	5	5	5
95	3	3	4	5		3	4	5	4
96	5	3	5	4	4	5	3	NA	3
97	1	3	4	4	2	2	1	NA	1
98	3	4	4	5	2	5	4	5	5
99	4	4	3	5		1	1	5	5

Respondent ID / Question ID	*3*	*1*	*9*	*2*	*4*	*5*	*6*	*7*	*8*
100	4	4	5	NA	4		5	NA	1
101	4	4	3	NA			NA	NA	1