Software Systems In-House Integration
– Observations concerning Architecture and Process

Rikard Land: Ph.D. Thesis

Draft as of: July 7, 2006 12:50
Abstract

Software evolution is a crucial activity for software organizations, and an increasingly important evolution activity is the integration of previously isolated systems. One goal of software integration is to increase the user value by combining the functionality of several systems, and another is to reduce functionality overlap. If the systems are completely owned and controlled in-house, there is an additional desire to rationalize the use of internal resources by decreasing the amount of software with essentially the same purpose. Situations requiring this type of systems integration are common, being the result of various organizational changes. This thesis builds up understanding of the problems associated with this type of in-house integration and provides guidelines concerning how to efficiently utilize the existing systems and the people in the organization to create a single system.

In this 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, and the software process perspective has led to the identification of important characteristics and practices of the integration process. These guidelines will assist future in-house integration efforts in making well-founded decisions timely and efficiently.

These 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 2.3 (page 26), 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 to being. Indoctrinated by the scientific paradigm, let me do this by listing the most important events that have contributed to this thesis in numbers:

1 supervisor (thanks Ivica!),

1 tool implementation (thanks to Mathias Alexandersson, Sebastien Bourgeois, Marko Burazin, 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 Ćurin, Marin Orlić, also to Igor Ćavrak for all collaboration),

20 published research papers and 4 technical reports (thanks to my fellow authors for a pleasant cooperation: Laurens Blankers, Jan Carlson, Ivica Crnković, Igor Ćavrak, 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 organisations),

31.5 years of life experience (too many people have been involved to be mentioned)

1,500 (estimated number) cups of tea, coffe 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 are 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, and the rest of my family)

Rikard Land, July 2006
# Table of Contents

Chapter 1. Introduction .................................................................11  
1.1 Scope and Assumptions.............................................................12  
1.2 Research Questions .................................................................13  
1.3 Research Phases and Methods ..................................................15  
1.4 Thesis Overview ....................................................................18  

Chapter 2. Research Results ......................................................21  
2.1 Process Model for In-House Integration ....................................22  
2.2 Practices ................................................................................24  
2.3 Architectural Analysis ...............................................................26  
2.4 Summary of Included Papers ..................................................28  

Chapter 3. Validity of the Research ..............................................31  
3.1 Research Traditions .................................................................31  
3.2 Relevant Research Methods ...................................................35  
3.3 Rigor and Validity in Each Research Phase ...............................40  
3.4 Overall External Validity .........................................................50  

Chapter 4. Related Work .............................................................51  
4.1 Software Evolution and Integration ..........................................51  
4.2 Software Architecture ............................................................61  
4.3 Processes and People ..............................................................68  

Chapter 5. Conclusions and Future Work .....................................73  

References ..................................................................................76  

Paper I .......................................................................................107  

Paper II .....................................................................................131  

Paper III .....................................................................................139
Paper IV.................................................................................................................................151
Paper V ......................................................................................................................................173
Paper VI.....................................................................................................................................197
Appendix A: Questionnaire Form and Data for Phase One ...............213
Appendix B: Interview Questions for Phase Three .........................229
Appendix C: Interview Questions for Phase Four ...............................233
Appendix D: Questionnaire Form for Phase Five ...............................237
Appendix E: Questionnaire Data for Phase Five .................................249
Chapter 1. Introduction

It is well known that successful software systems have to evolve to stay successful – as a consequence they are modified in various ways and released anew [230,288,291]. 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 [257] or (in the domain of enterprise information systems) through middleware [50]. 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 subject to little research, namely integration of systems which have overlapping functionality. For such overlapping systems developed and controlled in-house (i.e. within a single organization), the challenges involved in this kind of systems integration – although commonly occurring in practice – has been studied even less. I have (together with colleagues) labeled this challenge 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 scenarios that could cause this situation: the systems may initially have been built to address different problems in different parts of the organization, but have evolved and grown to include more and more functionality until there is a significant overlap and they are large enough to attract management’s attention. Other, more drastic, events include company

\(^1\) 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, 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, difficult to understand, and hard to change.
acquisitions and mergers, and other types of close collaborations between 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 marketing.

An increasing number of products contain software, and there is also an increasing trend of building and using software internally for use within a single organization. Reorganizations and company mergers are also a common phenomenon, which means that it becomes increasingly important to properly handle the situation of overlapping software systems. Although many organizations to date without doubt have encountered this challenge, and more or less successfully handled it, these experiences have – to my knowledge – not been collected systematically across organizations and published.

In this thesis, I present a sequence of research studies collecting experiences from 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 as well as more concrete practices – so that decisions can be made rapidly while being as well-founded as possible, and so that the cost and time of the implementation process is predictable. If some important activities are not performed, the decisions may be ill-founded and the integration late and costly, or never completed, and/or the integrated system may be of low quality.

- **Software Architecture.** The existing systems are arguably among the most important artifacts to study and evaluate. This evaluation needs to be done 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 profound impact, in particular data models and technologies used.

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 [148] and how to make the suggested processes and practices actually work [310]. Proper application of the theory and practice of management, business, (organizational) psychology, would arguably contribute greatly to the success or failure of an in-house integration effort.

1.2 Research Questions

The question for an organization faced with the in-house integration challenge is how to make as good decisions as possible, as rapidly as possible, which is what this thesis aims to answer.

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 efficiently (in terms of time and money) make the transition 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 evaluate properly and unambiguously. One measure would be the overall turnover and profit of the organization. However, from a scientific standpoint one would need to know exactly how
much the integration efforts contributed, including all indirect effects. Also, one should expect integration efforts to have a significant up-front cost, and it becomes problematic to define when to evaluate the economic result.

Nevertheless, I did not want to formulate a less interesting research question because it would be easier to answer. Aware of these limitations, I set out to pursue the question I believe would be the most interesting to find answers to, 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 can be found in existing organizations today?
2. **Evaluate Existing Practice.** What are the experiences of these organizations? In their own opinion, what mistakes did they do, 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

The type of study method depends on the research problem and the maturity of the research field. 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 aims at quantifying the subject studied. This is a general observation [302,335], which well describes the research of this thesis.

As described in the section on research questions (Section 1.2), the research has started with surveying the current state in organizations, together with their own evaluation of how successful they have been. These experiences have then been generalized into guidelines. According to this series of study questions, the research has progressed through five clearly distinguishable research phases, see Figure 1. 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. Experiences from more organizations were collected (phase three), which led to two follow-up studies: one studying Merge more closely (phase four), and one validating and quantifying the previous findings (phase five).

The rest of this section introduces each of the phases briefly with a paragraph each. The research method of each is described in depth in section

Figure 1. Research phases.
3.3 and the complete published results of each can be found in the appended papers and appendices.

**Phase One: Exploratory Case Study.** I had the opportunity to participate in an industrial project, where 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 to discontinue some of the existing system(s). Management was then to agree upon implementing one of these solutions. The data used is my own experiences as a participant, and a questionnaire which collected 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 [201,205,211]. Two of these are included in this thesis as Paper I and Paper III. These events were also discussed in the licentiate thesis\(^3\) [200].

**Phase Two: Literature Survey on Integration.** In phase one, it was difficult to position the case study in relation to existing literature. To investigate to what extent the case experiences were unique among existing research publications, a major literature survey was performed. Of course, relevant literature had been searched for and consulted both before and after, but in this phase a systematic search scheme was designed, where publications containing certain keywords were searched for in publication databases, book lists, etc. Many publications were discarded based on title and abstract, but many were screened, and many publications studied more thoroughly. Promising references were also inspected, until not much new was found. This literature survey resulted in one publication [206], 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 more cases with experience from in-house integration projects were actively sought for. There was had at this stage no developed theory, but asked various questions concerning the integration process, with a particular focus on technical characteristics of the systems. I studied nine such cases,

\[^3\] The Licentiate degree is a Swedish degree somewhere between a M.Sc. and a Ph.D. degree.
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 [203,208-210,214], both process related [203,208,214] and architecture related [203,209,210]. These were later combined and extended into one journal paper [207], 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 having the most variants and being the most difficult to implement successfully. Therefore, I 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). Based on this data, a method for rapidly exploring Merge alternatives has been devised. With the help of students, a prototype software tool to support the method has also been developed. This phase resulted in one conference publication [204] describing the method itself and one workshop publication [212] 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, one natural continuation was to design a study aimed at validating these. Also, 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. Therefore, a questionnaire was designed consisting of a number of questions with five-grade scales. The questionnaire was distributed to six of the previous cases plus two more. (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 [215] which is included as Paper IV. The questionnaire form and the collected data are reprinted in Appendices D and E.

\(^4\) Details about how we use this term can be found in section 2.1.
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.

![Figure 2. The concepts of the thesis and their relationships.](image)

The thesis is organized in the following way: there is a chapter or section 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 over the goals, research methods, and resulting papers of the five research phases. Chapter 1 surveys related work related to the thesis, and is divided into three main parts: integration, architecture, and processes. 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 76. 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 with earlier sections of the thesis.
Chapter 2. Research Results

This chapter describes a brief overview of the research results, since the details are found in the appended papers. Figure 3 provides a high-level overview of the results by describing the important elements of a proposed integration process and its various elements. There are two phases or subprocesses: a vision process (which ends in a decision) and an implementation process. Of these two, the thesis focuses on the vision process, which involves considerations of various strategies for the final system and their associated project plans. To be able to make a decision 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, 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 the results in the different papers to each other; therefore we 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 made. This chapter ends with section 2.4, where the included papers and contributions of each paper (in particular mine) are listed.
2.1 Process Model for In-House Integration

In-house integration is typically initiated by the senior management, as a result of 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, the participants of each, and the activities belonging to each (Paper II, section 1.2; Paper III, section 2).

Figure 3. The important elements of the proposed integration process.
The vision process leads to a decision that includes a high-level description of the future system both in terms of features (requirements) and design (architectural description), as well as a 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 executing 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 complete – 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 of 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 are built the way they are, while the considerations on 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 analyze in order to understand the consequences of each potential strategy (Paper II, section 8.2) – even when an outlined alternative lies somewhere between these idealized strategies, which is common (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 how well 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 a Merge, 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 disassembling the components; the same type of analyses (i.e. impact of retirement, compatibility, etc.) need to be made for these alternatives (Paper II, sections 4.1 and 4.3).

An implementation plan needs to be outlined for the selected strategy, considering available resources and what costs and risks would be acceptable (Paper I, sections 4.2 and 4.3; Paper II, sections 6 and 7). Depending on the strategy, this plan will look different. 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 a Merge, stepwise deliveries of the existing systems should be planned, thus enabling an Evolutionary Merge, and take into account the complexity of the parallel maintenance and evolution of the existing systems (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 Evolutionary Merge (Paper I, section 4.2; Paper II, sections 1.2, 2.2, and 8.1). A Merge also means more, and longer lasting, distributed development and a need for synchronization as well as a stronger, and longer lasting, tension between local and global prioritizations at each site (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 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 further studied by a questionnaire in research phase five (Paper IV, section 3.5).

During the vision process, two closely related practices were identified: to gather 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 collected overview
over the systems must be created, 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
requirements elicitation already done 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 as
the existing systems already do; 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 distinguish which
practice of timely decision and sufficient analysis is to be preferred 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). All these aspects are often 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
enough 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 have been used
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 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 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, so that (most of) the changes made to the existing
systems in order to achieve the long-term integration goal are also useful in
the short term; this will, to some extent, make the convergence accelerate on
its own (Paper II, section 6.2).

2.3 Architectural Analysis

The findings and understanding concerning architectural analysis have
evolved and been refined through all phases, from initial observations and
lessons learned [289,336] 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 to reuse 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 the components are annotated with some properties
of interest (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 components are annotated
with effort of change, it is possible to calculate a rough estimation 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 most 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 there are too large differences: structures, frameworks, and data models (Paper II, section 4.4). Based on the studied cases, there are strong indications that the structure of the systems must be very similar to make it possible to Merge them – or at least that it is otherwise not feasible in practice (Paper II, section 4.3). “Framework” should in this context 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 to make Merge feasible – however, an exact match is not 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 a Merge is practically infeasible (Paper II, section 4.4).

True at least for the cases studied, the systems to be integrated often exhibit 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 some 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
often makes 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, there are some standards within that particular domain, and the existing systems might even have some common ancestry due to previous collaborations, the systems are not unlikely to be similar enough for a Merge to be possible.

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 [289,336] 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 while making observations and reflections. My supervisor and coauthor was a valuable mentor, and both authors related the case study to existing research literature, and formulated general conclusions.


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 on the retirement of existing systems, other issues to evaluate, and observed practices. This paper builds on several earlier conference publications [203,208-210,214].

I was the main author; my contribution was to initiate and lead the study. Early design and analysis was done with the help of my supervisor and coauthor (as well as other colleagues, being co-authors of the earlier conference papers). During the writing process, led by myself, my supervisor and coauthor have given many suggestions and a lot of advice, and we have had many constructive discussions.


This paper describes the case study of phase one, focusing on overall process characteristics and some 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 and made observations and reflections. The coauthors aided in relating the case study to existing research literature and formulating general conclusions.


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 was to lead all phases of the study. The coauthors were involved in outlining the study, designing and distributing the questionnaire, analyzing the results, and writing the paper. Peter Thilenius stood for the expertise concerning questionnaire design and statistical analysis.

This paper is based on a follow-up study on a Merge case. The paper suggests a method how to explore various Merge alternatives, by making incompatibilities explicit, record decisions made, and guide the exploration based on the entered information. 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.


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

I was the main author; my contribution was to act as customer and steering group for a student group in a project course which implemented the tool. I invited one of the students to be coauthor; he made some further updates of the tool after the course had ended and participated in the writing process.
Chapter 3. Validity of the Research

Why should the results of this thesis be believed? 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 are achieved by systematic study, and claim some amount of external validity of the results, i.e. establishing the domain to which a study’s findings can be generalized.

In the research field of Software Engineering, several research traditions and methods meet. Here we find mathematical reasoning alongside studies of humans, technology, business, society, and their interaction. Quantitative studies share space 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” [336]. This chapter therefore starts 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 is achieved, the larger part of the chapter accounts in detail for how the research has been carried out (section 3.3). These accounts are synthesized in Section 3.4 which concludes this chapter by discussing to what extent the results are externally valid.

3.1 Research Traditions

There exist a number of research traditions, of which those most influential in shaping the field of Software Engineering are briefly related 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 are able to make 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 [289]. 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 [62, 297]. The commonsense inductive argument says that the more supporting data, the stronger supported the theory is. However, this standpoint is difficult to defend logically [62, 297], and an alternative is the notion of corroboration [297], 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 for relatively simple, quantifiable models. Also, observations are subject to e.g. measurement errors, insufficient measurement instruments, inappropriate use of these instruments, and not least, observations are always made by an observer with some predispositions [62, 75]. 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 [62]. Historically, this has caused numerous controversies between competing theories, where the proponents of each side disqualify the other’s observations and experimental settings [75]. 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 [289]. This is typical for social sciences and is common in Software Engineering when it comes to studies of the social and

---

\(^5\) Total objectivity may be a too idealistic view; however, the researcher should strive for 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 humble distinguishing “science” from “non-science” [62]. Taken somewhat to the extreme, these arguments has led to deconstructive and relativistic standpoints, according to which science is mainly a social activity (i.e. scientists have achieved a certain status), and there is consequently no such thing as validity.
psychological aspects of software, such as usability [271,361] or the introduction of a new process or method into a development team [187].

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 [289,378]. The notion of text can be extended beyond written texts to include speech, multimedia, or any event. 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 understanding of the artifact studied (see further discussion under 3.1.2 below).

Computer Science is largely founded on logics and mathematics, where there are no observations of an external world [359]; 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) [187,320].

Ethnography takes a cultural perspective [289], and has found its way into Software Engineering [310]. 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 [289,378]. Other traditions include the positivist and realist traditions, but these seem less influential in Software Engineering as their primary focus is the notions of reality and truth [378]; 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 on validity (e.g. conformance to a paradigm in normal science [75,196]) but are of no help for individual researchers or individual studies [62], 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 requires the question being studied to be well-understood and feasible 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 does not yet have a strong theory that guides the design of the study, when the context cannot be separated from the phenomenon being studied, and/or when people’s individual experiences of the phenomenon is as important as the phenomenon itself [289]. 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 [302]. Many study questions, not least in the field of Software Engineering, are several-faceted and thus need to include both quantitative measurements and qualitative data [289].

Four commonly referred to types of validity are: construct validity, internal validity, reliability (or conclusion validity), and external validity (or generalizability) [302,381,388]. (These are further discussed in section 3.3.) These are applicable to both quantitative and qualitative research, and especially the first three are closely connected with the traditional evaluation criteria for research such as researcher objectivity, systematic and rigorous procedures, and triangulation [289]. 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, where terms like probability and confidence have a well-defined mathematical meaning [ref]. External validity is argued for by showing that the prerequisites are fulfilled (i.e. the population is well defined, some appropriate sampling strategy has been chosen, etc.). Although this is to some extent also applicable to qualitative research, it has been argued that understanding is ultimately the only validation possible [247]. This understanding is to be judged by other researchers (and practitioners, I would like to add). 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 events heavily dependent on social and economical context (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.” [95] However, validation cannot be totally arbitrary; any claim needs to be strongly supported by data and the reasoning
that led to a certain conclusion [247]. I agree that “insight, untested and unsupported, is an insufficient guarantee of truth.” [309]

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 there is a theory being 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 one wants to make some sense out of seemingly chaotic data, after which a theory is formulated, and there is first a focus on gathering some support and only later on testing it through falsification attempts, or comparing to rival theories [62,336].
This section describes some common research methods and the circumstances under which they are suited, and the choices of research methods in the five phases are motivated.

3.2.1 The Case Study

For contemporary problems which cannot be properly studied outside its complex context – and where the complete context is not even known – the case study [388] 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 [388]. 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 strong 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 to learn from (an extreme case or illuminative case) [289,388]. Time and resource limitations might also prohibit more than one case to be studied. A revelatory case is one that the researcher only during (or after) the study understands the importance of, for example in characterizing a new research problem [388]. Often the results of case studies are reported as observations or lessons learned [289,336]. If a case study is planned so that a contemporary event is studied when it happens, it is possible to do the same measurements before and after the event – which is a benefit 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 – such as the topic of in-house integration.

The problem with case studies is that the complex context, in terms of many influencing (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) [187], but to be able to claim any wider external validity, it seems as the best one can do is to be rigorous in every way, and propose and evaluate several explanations – rival theories – to explain the results [289,388]. And as said 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 [29,95,247].
3.2.2 Grounded Theory Research

According to the grounded theory research method [347], 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, where 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 accommodate 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. There is a need for being 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 [266,268,351]. 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 [289].

Grounded theory should not be mistaken for free-range exploration with no predispositions at all; this is seldom the case for a researcher [302]. 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. In my opinion, the strength of the grounded theory method is that it codifies the element of an early qualitative study (when there is yet no theory to be tested) in that it emphasizes a constant interplay between data and theory [266,289].

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 rigor. For studies in social sciences, where the grounded theory method comes from, 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 that fit the traditional research criteria best (e.g. objectivity of researcher, systematic and rigorous procedures, validity of data, triangulation, reliability, external validity) [289].

3.2.3 The Experiment

The classical method for 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 put theories to strong tests 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 also found its way into Software Engineering [23,355,381,391]. For example, if one wants to assess whether the usage of a certain process is better (in some sense) than another, one could study a project group following the process and an equivalent project group following the other, and measure which was most successful. Large-scale complex phenomena, which are out of the control of the researcher, can be studied in a natural experiment [289]. 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 e.g. precisely formulating and analyzing concepts and ideas (see e.g. [1,6,77]). However, in Software Engineering the usefulness or feasibility of something (which must be studied with some other method) is equally important as its formal correctness.

3.2.5 Construction

Also common in Software Engineering research is the construction of software as a proof of some concepts; often this software is in the form of a tool supporting a process [89,90,101,172,174,241,314,315]. Seen in
isolation, the scientific value can only be to prove that building this kind of software is possible (which may indeed sometimes be an achievement) [336]. 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 concerns a contemporary, complex phenomenon, research is largely based on case studies. In my first research phase, 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 the case study would later be used as an illustration (it is thus a revelatory case). There was no particular theory or proposition, and the only reasonable thing to do was to collect further experiences (with a chosen 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 that had clearly chosen and successfully implemented a Merge (although implementation is not finished 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. Especially 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 had a theory developed enough to carry out an experiment, neither have there been fine enough instruments for measuring the outcome.

Data has been collected through project participation, direct observations, interviews, and questionnaires. In my literature studies, I have aimed at being as rigorous and systematic as possible, including 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 [95,289,302,388], 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 [388]. For interviews and questionnaires, construct validity also means that the researcher must also avoid leading or ambiguous questions [302].

- **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 [388].

- **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 [247,302]. Descriptions of data must be accurate, which can be ensured by introducing a review step where informants review e.g. copied out interview notes [302]. The researcher must also be open to different interpretations and theories, and avoid being predisposed to specific interpretations [247,302]. To increase the internal validity, there are several types of triangulation that should be employed [95,388]: 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 [205] and as Paper III is a description of the process used.
In addition, we reported how the IEEE standard 1471-2000 [156] was used in the discussions on architecture [201].

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 [388] 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 [388].

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: Literature Survey on Integration
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 [185,344]. 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. Internal validity is satisfactory, as the distance is very small between the

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).
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 [66], ELIN@Mälardalen (a search engine at my university, which searches several databases simultaneously), Google [120], IEEE Xplore [155], Kluwer and Springer journals [341]. The keywords used in the search were: “integration”, “interoperability”, “reuse”, and “merge”. The concrete result of this phase was a conference paper [206] 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, 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 [388] 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 [213]). 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 [213]), and the rest of this section presents the research method, in particular how threats to the different types of validity were addressed. The text in this section is heavily based on the technical report [213].

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 [302], which I have tried to follow. In some cases, the interviewees offered documents of different kinds (refer to the technical report [213] for more details).

Construct Validity

Multiple sources of evidence has 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 [388] 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 [95,388] 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 [388] (or analytically generalized [302]) 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 [388], which was carefully selected as an extreme case in a good sense, illustrating how a Merge 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 [202]. 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 [202].

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

Reliability
Yin’s [388] 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 [95,388] 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
Some 2-4 A4 pages will be reused from the technical report supplementing Paper IV.

Refer to Appendix D and Appendix E.

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 also with 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 available advice concerning how to implement it rigorously and systematically has been followed as far as practically possible. The largest limitation of the result is of practical nature, in the number of cases studied and the number 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 how sizes of organization, system domains, and other factors affect these results.
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, in order to understand commonalities and differences. 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 over 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 [200] and integration in [206].

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 [230] (although it is sometimes used e.g. for programs modifying themselves [ref] or evolutionary programming techniques such as genetic algorithms [20,258]). 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 [52,229,230,291]. The term “maintenance” usually means making relatively small changes, and can be classified into perfective maintenance, corrective maintenance, preventive maintenance, etc. [153,286,294,340]. There is, however, not a clear border between these, in spite of efforts to define them and create different process models for these [167]. 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 [53] of the system will be violated and deteriorate (or “erode” or “degrade”) [24,41,163,288,338,367]. Complexity will increase unless work is done to reduce it [229]. 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 [366]. The long term optimum should be a proper balance can be found so that maintainability is maintained [199,299], but there are still many challenges to be addressed [256]. When modifying a system it is important to understand the rationale behind a system’s design in order to avoid design deterioration [49,225]. Maintenance is not only a post-deployment activity but should be carefully planned already during system’s development [170,294]. 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. [282]; there are even approaches to update systems in runtime (based on its componentized architecture) [283].

The software organization itself, including its tools and processes, affect how well it performs in maintaining and evolving its software [153,168,186,233,299]. Not only the actual software is subject to maintenance; maintenance could also involve other artifacts such as the software’s documentation [3,170,197]. There are process and maturity models addressing maintenance [13,158,169]. 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) [63,168,186,233].

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 [24,31,151,393]). 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 [3,14,74,217,275,276,375,393], the Halstead measures [129,326] and cyclomatic complexity [3,129,276,325]. The Maintainability Index (MI) aggregates several of these measures into one [276,327]. 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”) [30,107,123,138,162,217]. These are static measures; tracking changes in these between versions could be a way to monitor software deterioration [14,74,199,227,228,300,301,357].

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 [230]. For example, web-enabling a system [164], moving from batch execution to real-time service [216] and/or componentizing an existing system [140,164,254] represent such major leaps – as do in-house integration.

There is literature how evolution has been addressed at the architectural level [65,165,252], not least in the form of architecture level evaluation methods based on change scenarios [24,31,69,176,190,240] (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 [25,45,61,125,316,357].

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 [151] 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 [374,379]. 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” [151] or “a point at which independent systems or components meet and act or communicate with each other” [122,311]. 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 [111,114] 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 [43,105,106] – structures assuming to be in charge of the high-level design decisions – are used simultaneously [246]. A survey over the field of architectural mismatch gives by hand the research is relatively immature [35]: there is more research to be found on how to detect mismatches [32,88,89,101,385] than how to solve them, and these approaches are typically not validated in a real-life environment. Some interaction mechanisms could be deferred until integration [92]. Some design patterns [112] and architectural patterns [57,109,184,318] address reuse, maintenance and evolution, but there is to my knowledge only little research on design patterns facilitating integration [181,237,387].

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 [48]. Without a strong vendor, architectures may still be the integration enabler by means of a standard reference architecture (reached through common consensus) [257,331]. Component models such as CORBA, COM, J2EE, .NET, embeds architectural decisions and may be considered middleware architectures facilitating interoperability [389].

To be able to integrate systems, the systems’ views of the data – i.e. their data models, taxonomies, or ontologies [126] must also be integrated, an undertaking not so trivial [149,200,277,295,346]. Geographic Information
Systems (GIS) [56,236] is one significant example of a domain where ontology integration has attracted attention [71,274,333,371]. 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 [19,79,136,312,350] 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 [82,136,350]. There is a strong focus on explicit interfaces [19,82,136,257,330,350,373], which – to enable true interoperability – must be built according to common rules. Component models are such sets of rules, supported and enforced by component technologies, such as CORBA [339] (a standard from OMG [278] with several vendor-specific implementations), Java 2 Enterprise Edition (J2EE) [261,303] (originating from SUN Microsystems), and COM [46] and .NET [353] (from Microsoft).

The existence of such common integration rules paves the way 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 [257,373]. Even when a system is completely developed in-house, a component-based approach may be chosen: a product line approach [68] means that there is a strategy for internal development of components to be reused in different products. This approach poses new challenges to the software community, e.g. mechanisms for variability to enable evolution of the products of the product line [349], new and stronger mechanisms to track changes to prevent the common assets from degradation [163,348], configuration management to control product derivation and evolution at the same time [362,363], and how to use stakeholder scenarios to evaluate the suitability of a product line architecture [190]. 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 [44][ref] (this is further discussed in section 4.2.2).

Integration at the function call level is relatively straightforward, but the Interface Definition Languages (IDLs) of the current component models can only achieve syntactic interoperability [171,380], which is not enough to
make two components interact as desired [374,389]. To ensure true interoperability between systems or components, the semantics must be specified as well [135,242,272,374].

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 configuration management techniques [218,220,362-364]. Using third party components in a long-lived system creates an undesired dependency regarding maintenance, updates, error corrections, etc. Also, it may be very costly to exchange one component to one that is similar. There is not yet a standardized way of certifying component quality and behavior although there is research on how it could be achieved [82,141,219].

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 [257,330]. 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 [264].)

The notion of open standards is widespread. Major organizations for software standards are ANSI [12], IEEE [154], and ISO [157]. 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 [128,269]. Other fields in similar contexts, where systems from different vendors need to interoperate and exchange information are Geographic Information Systems (GIS) [55,71,182,224,274,352,358] and hypermedia [10,11,87,149,259,376] 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 [284], mathematical computations [223], and photo archives [192]. Interoperability through standardized interfaces is also a concern of software agents [224,382]. 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 [224].
XML [131,384] has become a popular encoding language which may be a common denominator of systems and used for integration [11,64,91,384]. 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 [183]. Conformance to a standard is in practice not enough to ensure interoperability between two implementations [54,231,248].

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 [103,304]. 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 [284].

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 [232]. 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 [47,83,226,273] systems such as SAP R/3 [313], *Product Data Management* (PDM) and *Software Configuration Management* (SCM) systems [80], and electronic business systems e.g. for business to business relationships, B2B [235,386]. As enterprises need to streamline their processes to be competitive there is a need for integrating these systems [132,222] 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 [127]. Since building unique interfaces between each pair of systems that needs to communicate is not cost efficient [103], numerous systematic approaches are used to enable a more structured integration of enterprise applications [132,226,237,346]. These are collectively called Enterprise Application Integration (EAI) [83,165,234,235,306] and include activities such as data mining and reverse engineering [5] and content integration [346] (to understand the existing data and systems), migration [51] (to get rid of the most problematic technologies and solutions), using a common messaging middleware [50,234,235,250,306,389] (of which there are many commercial solutions), and encapsulating and wrapping legacy systems in a component-based approach [317]. The market for application integration and middleware (AIM) is estimated to $6.4 billion worldwide in 2005 and is expected to continue growing [78].

EAI requires a high degree of commitment, coordination, and upfront investments [226]. EAI may break down when integration occurs between enterprises, when data is operational rather than historical, and more unstructured data need to be integrated [346]. And the integration problem continues: as systems being integrated use different (not fully compatible) commercial technologies, the need arises to integrate the integration technologies [121].

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 [72,102,221,311]. In this context, the system is designed top-down, followed by implementation of the various parts (and possibly acquirement of existing components), which are then integrated [102]. 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 [249]; this will also give a hint of the emergent system properties. Interface specification and coordination are important activities [72,102,122], and a systematic implementation of the component-based paradigm enables parallel development and (hopefully) smooth integration [221].
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 version management system [33,255]. 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 [255]. By narrowing down the application domain to e.g. a specific programming language, it becomes possible to perform a syntactic merge [255]; also resolving semantic conflicts is more difficult, and it is in general an undecidable problem [359][ref]. The parallel development branches may be refactored differently, which gives rise to structural conflicts which need to be resolved in order to enable a merge at this higher level; this however seem to be a largely open research area [255,365].

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 [15,255,293]. (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” is only 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 [354]. 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. Related is the observation that building a system using two object-oriented frameworks simultaneously gives integration problems due to conflicting assumptions [246].

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 [35]. 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 [121,135,165,226,255,285,343,374], 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 [200].
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 [53,98,287]. Object-oriented analysis and design partly addressed this as systems grew larger [38,160,307]. The foundations of the field of software architecture were laid [1,94,116,292,337] as the first architectural languages were designed [238], the need for views [194], the rise of the pattern community [57,112], special issues of journals [152], and the first books [57,338].

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 [25,57,156,338], 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 [334]. However, this view is limited; there is for example limited value in visualizing code structure without knowing the intentions behind the design [25,39,45,61,125,316,357]. Some texts emphasize the rationale of the architectural decisions [70,156,292] which is important for maintainers will arguably be able to perform changes efficient and without violating the conceptual integrity of the system [49,225] (cf. the discussion in software deterioration in section 4.1.1). Some texts focus on the architecture as 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) [110,195]. Different stakeholders have different needs of an architecture (and its description) that should be addressed [25,70,156], and the business and organizational context of a system could also be considered an essential part of the architecture [279,390,394]. Along this line, the role of the architect is perhaps as important to discuss as the architecture [99,188,245,265,332], and there are associations of architects [150,383].

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 [25,42,65,195,290]. 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 [25,69,70,156]. Architectural descriptions can also be analyzed, which makes it possible to evaluate alternative architectures before a system is built [25,67,69,175,177,178] (see section 4.2.5).

There is a correlation between the structure of an organization and that of its software [76,132]. The integration may occur at different levels, ranging from data and application to the more difficult levels: business processes and humans [296]. 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 [390,394]. Another enterprise information systems framework is “The Open Group Architectural Framework” (TOGAF) [279].

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 [44]. 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 [44]: CORBA [339] (a standard from OMG [278] with several vendor-specific implementations) is designed for distributed real-time and performance-critical applications, Java 2 Enterprise Edition (J2EE) [261,303] for distributed enterprise systems, and COM [46] and .NET [353] address the desktop domain. These component models have been compared from this architecting point of view [96,104,389].

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 [141-145,219,262,342].

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” [53]. 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” [156]. When discussing systems in general, the appropriate term to use is viewpoint [156] or viewtype [70], 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 [194]. There are slightly varying names of essentially the same viewpoints, such as the suggested conceptual view, execution view, module view, and code view [57,70,147]. There are also suggestions of additional views that could be useful in some cases, such as an architectonic viewpoint [243] and a build-time view [356]. There is research on how to formally relate elements of different viewtypes [137,263,392]. 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 [156].

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 [238]. The Carnegie Mellon University has constructed several ADLs as part of their research, such as UniCon [338], Aesop [113], 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. [253,321,324] for further references. Acme, developed by a team at Carnegie Mellon University is designed to be an interchange format between other languages and tools [115], but should possibly be considered as a new ADL in its own right [86]. The Architecture
Description Markup Language (ADML) is an XML representation of Acme with some extensions and transparent extensibility [281].

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 [368,369]. The Fundamental Modeling Concepts (FMC) [134,179,180] 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 [125,133].

The Unified Modeling Language (UML) originated from object-oriented design and modeling [40,147,360], 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 [73] it became the de facto language used in industry to model architectures [189,193,253]. UML 2.0 [280] provides more capabilities for modeling architectures, and it provides several extension mechanisms that may be used to support architectural constructs [40,251,308]. It could still be confusing to use the same notation for different levels of abstraction [146].

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 [57,112] as well as patterns for various domains, such as distributed systems [318], resource management [184], and enterprise systems [109]. 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 [39]. As with views, styles abstract away certain elements and emphasize others [70], 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 [24,70,337,338,372]; a simple compiler is typically considered a typical example of a pipe-and-filter architecture [4,338]. A blackboard (or
repository) architecture draws the attention to the data in the system [24,337,338,372]. In a client-server architecture [24,41,329,337,338,372], 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 [24,41,337,338,372]. 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 [25,70,328]. Object-orientation has also been discussed as an architectural style [24,41,338], 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 [338,372]. 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 [57,109,112,184,318].

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 [25,67,69,175,177,178]. 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 [7,97,100], or consistent implementation of a style [1], or to execute or simulate it to assess other properties [6,7,21,22,238].

There are techniques to evaluate system performance based on architectural descriptions [97,100]. Certain qualities may also be assessed through simulation of an architectural description [238] 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) [25,239,260,319,377].

Another type of analyses should be made by involving the system stakeholders [24,40,41,177,178,191,305]. The Software Architecture Analysis Method (SAAM) uses stakeholder-generated scenarios to compare the quality properties of alternative architecture designs [24,69,174,175]. 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 [69,69,178]. 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 [69]. 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 [41]. 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 [69].

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 [69,176-178,198]. 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 [24,177,178].

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 labelled “framework” (in a different meaning than object oriented frameworks [166] and component based frameworks [82]).

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 [35] (cf. the discussion in section 4.1.2). Our studies has 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 [117,118,159,244,298,340,370]. 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 [298,340,370]. 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 [298], and conversely: a well modularized architecture may be a requirement e.g. when development teams are geographically distributed [59,173]. There is little room for evolution in the waterfall model; errors introduced in one phase are not found until its corresponding verification and validation phase (often illustrated in a “V”-shaped diagram) [118,244,298,340,370] – 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 [ref]. 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 [26,28,36,249,345,370]. With short enough iterations, or small enough increments, early feedback is provided as of potential integration problems as well as system properties [221,249].

There are processes with strong roots in a particular technology, most notably perhaps being object-oriented development [38,160,307]. 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 [118,195]. 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 [18,19,34,35,79,81,161,267]. 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 [17,35,85], means that the development process must be radically different from the top-down approach, and involve e.g. prototyping using potential components [34,35,161]. The recent trend of rapid development and agile processes emphasize the need for customer
involvement and continuously adapting to new requirements [26-28,249,345] (and also, we can note in the context of this thesis, continuous integration of work products [249]).

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 [72,102,158,159]. An overall goal of these models is to make projects and their outcomes predictable; actually the term “software production” has been used instead of “development” [118]. 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 [72,158].

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 [124,322]. The Six Sigma approach originated in hardware manufacturing, which focuses on continuous measurement and improvement of defect rates, is also being applied [323]. The reviewed process models and standards are typically used to guide such process improvement initiatives, e.g. by moving up maturity levels [13,72,158,169].

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 [58,270]. One can also argue that these models treat people as parts of a production machine, which suffocates creativity within an organization [93]. 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” [102].
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 [93,249]. Learning should be promoted and the staff’s skills continuously improved [84,102]. 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 [249].

To become productive, stakeholder commitment to a project is essential for success [2,72]. 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 [249,340,345].

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 [72,244,298,340,370]. Defined goals, planning and monitoring are essential for project success, preferably with quantified goals and metrics [72,119]. Project risks must be analyzed and addressed, early in a project and throughout [72,102,117,298,340].

Whatever work products are produced, verification and validation of these are a strongly advocated means of quality assurance [72,102,118,244,298,340,370]. 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 [72,117,130,249,340].

The agile movement seem to advocate a less formal requirements engineering, and instead focus on continuous customer involvement [26-28,345]. This is partly explained by their typical different application domains and project sizes [37]: the more heavy-weight processes address large-scale, critical software [72,159], agile methods seem to be most suited for small-scale projects of non-critical software [ref]. 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) [26-28,108]. It has been suggested that a system’s architecture plays an important role in all life cycle phases of a system [25,42,195,290].
As organizations become global, they face the challenge of employing distributed processes, i.e. where the team members are geographically dispersed [59,139,173]. 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 [59,60,148]. Meeting in person regularly is essential, and more formalized legal agreements are advised [59].

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 provide guidance for future in-house integration projects in order to build up a systematic approach that will lead to a more efficient and predictable process. To achieve this, we have surveyed, evaluated, and generalized the existing practice in a number of organizations, which has led to the formulation of a process model for in-house integration. The thesis focuses on the early vision process, which could and should be carried out relatively rapidly while ensuring enough coverage of the most important considerations.

One such consideration is the existing systems’ architectures and the characterization of their compatibility. If the systems are not sufficiently similar with regards to structure, data models, and frameworks (including technologies used), combining components from them is practically infeasible. Similarities are not that uncommon, however, and there are a few indicators that suggest a certain amount of compatibility: the presence of domain standards often leads to similar solutions, and systems that were built in the same time period also often exhibit similarities. For many other considerations, there is no objective way to find a solution; rather, the crucial activity is 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 somewhat unique is that each stakeholder initially knows only his or her own system well, and knows the other existing systems very little, which means that the existing systems must be evaluated and alternative integration solutions must be created in a neutral way. Practices must be employed that ensures efficiency and objectivity, although cooperation is not necessarily in all individuals’ interest in an in-house integration situation.

It is possible to integrate the process elements proposed in this thesis with most prevailing processes in an organization, whether they are plan-driven, agile, or ad-hoc. The guidance of 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
organizations are more familiar with: 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 for in-house integration in this phase as well. Several processes will run largely in parallel, and they must be properly synchronized and managed. This becomes more crucial 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, managing distributed teams brings a set of challenges on its own.

The research has been carried out systematically and rigorously, which makes the research results reliable. The main limitations of general validity are the relatively few number of cases that have been studied and the bias towards Swedish and western organizations and people; this bias can be discerned in the selection of cases studied as well as in the (partly unconscious) mindset of the researcher (me). It should also be remembered that already at the outset 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 once (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 notional cultures – and more cases from the domains and cultures already represented – the results will be stronger. I also welcome studies of this topic from other viewpoints, such as organizational and decision making psychology and the cultural aspects.

At a more detailed level, there are a number of loose threads that I think would be challenging and interesting to pursue further. First, the observations concerning how to elicit and document requirements of a future system was in my opinion very interesting, and this would be worth to study further as a topic on its own. Second, architectural patterns and styles could be an additional indicator that the systems are similar enough for a Merge to be practically possible. Third, the merge method and tool need to be evaluated for usefulness in realistic cases. Also, their further development should consider other viewpoints or languages than the simple module viewpoint currently implemented; 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.

If the research topic is relevant, which I believe it is, and has not been studied systematically and published publicly, which it has not, and the answers are non-trivial and attained through sound research, which I claim them to be, the contribution of the proposed thesis is significant. That is, as the title promises, the thesis gives substantial guidance for a systematic engineering process for in-house software systems integration and merge.
References


[174] 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.


International Conference Information Technology Interfaces (ITI), IEEE, 2005.


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

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?

Architecture-centered software development is a well-established strategy [2,3,13,21]. We have experienced the architecture of a system as an appropriate starting point around which to concentrate integration activities. One common experience is that integration is more complex and costly than first expected due to “architectural mismatches” [10,11], and this problem should be addressed at the architectural level. It also seems possible that some architectural analysis techniques used during new development could also be applicable during system evolution and integration. In this paper we show the extent to which an architecture-centric approach can be used during system evolution and integration, and how accurate and relevant the result of such an architecture-based analysis is.

Our aim has been to present our experiences from a case study in which three software systems were to be integrated after a company merger. We have monitored the decision process, and the actual integration has just begun. The activities were focused around the systems’ architectures. We describe the three integration approaches that were discerned and discussed, how architectural descriptions of the two most interesting were developed and analyzed and the decisions taken for the development project. Further we analyze the proposed solutions showing the strong and weak sides of the architectural strategy as such.

The rest of this paper is organized as follows. Section 2 provides the background of our case study, section 3 discusses four integration approaches, and section 4 uses the case study to elaborate on architectural analyses possible during system integration. Section 5 describes related
work, and section 6 concludes the paper and suggests directions for future research.

2. Introducing the Case Study

The case study concerns a large North American industrial enterprise with thousands of employees that acquired a smaller (approximately 800 employees) European company operating in the same business area. Software, mainly developed in-house, is used for simulations and management of simulation data, i.e. as tools for development and production of other products. The functionality of the software developed in the two organizations prior to the merger was found to overlap to some extent, and three systems suitable for integration were identified. A project was launched with the aim of arriving at a decision on strategic principles for the integration, based on the proposed architecture for the integrated system. This was the first major collaboration between the two previously separate software departments.

Figure 1 describes the existing systems’ architectures in a simplified manner in a high-level diagram combining an execution view of the system with the code view [2,7,13,16]. The sizes of the rectangles indicate the relative sizes of the components of the systems (as measured in lines of code). One system uses a proprietary object-oriented database, implemented as files accessed through library functions, while the other two systems, which were developed at the same site, share data in a common commercial relational database executing as a database server. The most modern system is built with three-tier architecture in Java 2 Enterprise Edition (J2EE), while the two older systems are developed to run in a Unix environment with only a thin X Windows client displaying the user interface (the “thin” client is denoted by a rectangle with zero height in the figure). These are written mostly in Tcl and C++, and C++ with the use of Motif. The “Tcl/C++ system” contains ~350 KLOC (thousands of lines of code), the “C++/Motif system” 140 KLOC, and the “Java system” 90 KLOC.
3. Integration Approaches

When developing architectures of new systems, the main goal is to achieve the functionality and quality properties of the system in accordance with the specified requirements and identified constraints. When, however, existing systems are to be integrated, there may be many more constraints to be considered: backward compatibility requirements, existing procedures in the organization, possible incompatibility between the systems, partial overlap of functionality, etc. Similarly, the integrated system is basically required to provide the same functionality as the separate systems did previously, but also, for example, to ensure data consistency and enable automation of certain tasks previously performed manually. When developing new software, it is possible to design a system that is conceptually integrated [5].
(i.e. conforms to a coherent set of design ideas), but this is typically not possible when integrating software since the existing software may have been built with different design concepts [11]. Another problem is how to deal with the existing systems during the integration phase (and even long after, if they have been delivered and are subject to long-term commitments). This problem becomes more complex the more calendar-time the integration will take as there is a pronounced tradeoff between costs in the short term and in the long term when different integration solutions have different maintainability characteristics. For example, there is an opportunity to replace older with more recent technologies to secure the system usability for the future. Scenarios possible if the systems are not integrated should also be considered.

In the analysis and decision process we have discerned four integration approaches or “levels” with different characteristics. They are:

- **Interoperability through import and export facilities.** The simplest form of using services between tools is to obtain interoperability by importing/exporting data and providing services. The data could either be transferred manually when data is needed, or automatically. To some extent, this could be done without modifying existing systems (e.g. if there is a known API or it is possible to access data directly from the data sources), and if source code is available it is possible to add these types of facilities. This approach would allow information to flow between the systems, which would give users a limited amount of increased value. It would be difficult to achieve an integrated and seamless process, as some data could be generated by a particular tool not necessarily capable of automatic execution. Moreover, there would be problems of data inconsistency.

- **Enterprise Application Integration (EAI).** Many systems used inside a company are acquired rather than built, and it is not an option to modify them. Such systems are used within a company, as opposed to the software products a company not only uses but also manufactures and installs at customers’ sites. Integrating such enterprise software systems involve using and building wrappers, adapters, or other types of connectors. In such a resulting “loose” integration the system components operate independently of each other and may store data in their own repository. Depending on the situation, EAI can be based on component technologies such as COM or CORBA, while in other cases EAI is enabled through import and export interfaces (as described in
previous bullet). Well-specified interfaces and intercommunication services (middleware) often play a crucial role in this type of integration.

- **Integration on data level.** By sharing data e.g. through the use of a common database, the users will benefit from access to more information. Since the systems store complementary information about the same data items; the information will be consistent, coherent and correct. However, it would presumably require more effort to reach there: a common data model must be defined and implemented and the existing systems must be modified to use this database. If this is done carefully, maintenance costs could be decreased since there is only one database to be maintained and there are opportunities to coordinate certain maintenance tasks. On the other hand, maintenance becomes more complex since the database must be compatible with three systems (which are possibly released in new versions independently). Also data integration may have an impact on code change, due to possible data inconsistencies or duplicated information.

- **Integration on source code level.** By “merging” source code, the users would experience one homogeneous system in which similar tasks are performed in the same way and there would be only one database (the commercial database used today by the C++/Motif system and the Java system). Future maintenance costs can be decreased since it would be conceptually integrated, and presumably the total number of lines of code, programming languages, third-party software and technologies used will decrease. Most probably the code integration would require integration of data.

Interoperability through import and export facilities is the most common way of beginning an integration initiative [8]. It is the fastest way to achieve (a limited amount of) increased functionality and it includes the lowest risk of all alternatives, which is the reason why managers usually adopt this approach. In a combination with a loose integration (EAI) it can provide a flexible and smooth integration process of transition: the import/export facilities can be successively replaced by communicating components and more and more integrated repositories. Of course, this approach has its disadvantages – in total it will arguably require more effort, and the final solution may technically not be as optimized as the results of the “data level” or “code level” approaches. This of course depends on the goals of the integration.

Which integration approach to use in a particular context depends not only the objective of the integration, but also e.g. the organizational context and
whether source code is available or not. For example, is the goal to produce an integrated product for the market, or is the system to be used only in-house? Is integration of software a result of a company merger? Is integration expected to decrease maintenance costs or to increase the value for users (or both)? Who owns the source code? Can the systems to be integrated be expected to be released in subsequent versions by (other) independent vendors? Is modifying source code an option, considering both its availability and possible legal restrictions? Business constraints also limit the possibilities – the resources are limited and time to market an important concern. One must also consider the risks associated with each alternative, meaning the probability of overrunning budget and/or schedule or not succeed with the integration. The risk parameters include not only those related to technical problems, but also those associated with the collaboration of two software development departments which had previously belonged to different companies and only recently began collaborating.

The project team of the case study intuitively felt that the benefits and the cost of implementation, the time to delivery, and the risk of the integration approaches described above should be related roughly as shown in Figure 2. The diagram is very simplistic assuming there is only one “benefit” dimension, but as mentioned earlier there may be different types of goals for integration, such as increased usability or decreased maintenance costs. EAI was never explicitly considered as a separate approach during the case study and is therefore omitted from the figure.

![Figure 2: Expected relations between risk, cost, and time to delivery.](image)
4. Development of Integration Alternatives

Developers from the two sites met and analyzed the existing systems at the architectural level, and then developed and analyzed two integration alternatives. The developers had architected, implemented and/or maintained the existing systems and were thus very experienced in the design rationale of the systems and the technologies used therein. The architectural alternatives were then handed over to management to decide which alternative should be used. The integration process was based on IEEE standard 1471-2000 [14] and is described in more detail in [17,18].

The “import/export level” interoperability was not discussed in any depth since it was apparent that more benefits were desired than could be expected with this approach. Instead, the software developers/architects tried the other approaches to integration, by conceptually combining the source code components of the existing system in different ways. The existing documentation had first to be improved by e.g. using the same notation (UML) and the same sets of architectural views (a code view and an execution view were considered sufficient) to make them easy to merge [18]. Each diagram contained about ten components, sufficient to permit the kind of reasoning that will be described. By annotating the existing components with associated effort, number of lines of code, language, technologies, and third-party software used, the developers could reason about how well the components would fit together. During the development of alternatives, statements about the quality properties of the integrated system such as performance and scalability were based on the characteristics of the existing systems. Patterns known to have caused deficiencies and strengths in the existing systems in these respects made it possible to evaluate and discard working alternatives rapidly. The developers had a list of such concerns, to ensure that all those of importance were addressed. The process of developing and refining alternatives and analyzing them was more iterative than is reflected in the present paper where we only present two remaining alternatives and the analyses of three specific concerns in more detail (sections 4.1 through 4.3).

The two remaining main alternatives conformed well to the “data level” and the “code level” integration approaches. Both these alternatives would necessarily need a common data model and shared data storage. From there, the two different levels of integration would require different types of actions: for “data level” integration, the existing systems would need to be modified due to changes in the data model, and for “code level” integration, much of the existing functionality would need to be rewritten in Java; see
Figure 3. In reality, these descriptions were more detailed than the figure suggests; About ten components were used in each of the same two views for describing the existing systems, a code view and an execution view.

Figure 3. The two main integration alternatives.

Architectural descriptions such as these make it possible to reason about several properties of the resulting integrated system.
4.1 Future Maintainability

The following factors were considered in the case study to be able to compare the future maintenance costs of the integration alternatives:

- **Technologies used.** The number of technologies used in the integrated system arguably tells something about its complexity. By technologies we more specifically mean the following: programming languages, development tools (such as code generators and environments), third-party software packages used in runtime, and interaction protocols. Too many such technologies will presumably create maintenance difficulties since maintaining staff needs to master a large number of languages and specific products and technologies, but at the same time tools and third-party software should of course be used whenever possible to increase efficiency. A reasonable number must therefore be estimated in any specific case. In our case study, the total number of languages and technologies used in the “code level” alternative would be reduced to 6 to 8 languages instead of the 11 found in the existing system combined, a number which would be preserved in the “data level” alternative. The number of third-party packages providing approximately the same functionality could be reduced from 9 to 5, and two other technologies would also become superfluous.

- **LOC.** The total number of lines of code (LOC) has been suggested as a measure of maintainability; it is e.g. part of the Maintainability Index (MI) [20,23]. In the case study, the total number of lines of code would be considerably less with the “code level” alternative. No numbers were estimated, but while the “code level” alternative would mean that code was merged and the number of lines of code would be less than today, the “data level” alternative would rather raise the need of duplicating more functionality in the long term.

- **Conceptual integrity.** Although a system commonly implements several architectural styles at the same time – “heterogeneous systems” [2] – this should come as a result of a conscious decision rather than fortuitously for the architecture to be conceptually integrated [5]. In the case study, it was clear, by considering the overall architectural styles of the systems, that the “data level” alternative involved three styles in parallel while the “code level” would reflect a single set of design ideas.

It might seem surprising that in the case study, in the “code level” integration alternative, the server is written totally in Java. Would it not be possible to pursue the EAI approach and produce a loosely integrated
solution, involving the reuse of existing parts written e.g. in C++? With the platform already in use, J2EE, it would be possible to write wrappers that “componentized” different parts of the legacy code. This was considered, and, by iteration the architectural description of this alternative was modified and analyzed with respect to the cost of implementation. Based on these estimates, all solutions involving wrappers and componentization were ultimately discarded and only the two alternatives already presented remained.

Whether to use Java or Tcl in the client for the “code level” alternative was the subject of discussion. Much more user interface code was available in the Tcl/C++ system than in the Java system which was preferable for other reasons. The pros and cons of each alternative were hard to quantify, and eventually this became a question of cost, left to the management to decide.

4.2 Cost Estimation

Estimating the cost of implementing an integrated system based on an architectural description is fairly straightforward. Based on previous experience, developers could estimate the effort associated with each component, considering whether it will remain unmodified, be modified, rewritten, or totally new in the integrated system. Clearly, the outcome of this type of estimation is no better than the estimations for individual components. The advantage of estimation at the component level is that it is easier to grasp, understand, and (we argue) estimate costs for smaller units than for the system as a whole.

This estimation is fairly informal and mainly based on experience, but it can be considered reasonable. First, the developers in the case study were very experienced in the existing systems and software development, second, the developers themselves agreed on the numbers, third, these numbers were higher than the management had expected (implying it not being overly optimistic/unrealistic), fourth, management explicitly asked the developers during the development of the alternatives to find cheaper (and faster) alternatives, something they were unable to do – the only alternative according to them would be the import and export facilities (for the interoperability approach). When summing the effort associated with all components in each alternative the developers found (partly to their surprise) that the implementation costs would be the same for both alternatives (the total estimated times differed by only 5%, which is negligible for such early, relatively rough estimations). This was true for the variant of the “code
level” alternative if Tcl was chosen for the client part - using Java would require more resources. The apparently high cost of the “data level” alternative was due to the definition of a common data model, and in the case of the Tcl/C++ system the use of a new database (a commercial relational database instead of an object-oriented proprietary database). These changes would ripple through the data access layer, the classes modeling the items in the database, and to a limited extent the user interface. Since the total number of lines of code is much greater than the estimated number of lines of code in the “code level” integration alternative, the apparently lower cost of modifying code instead of rewriting it would be nullified by the larger number of lines of code. It would also be necessary to write some new components in two languages.

Bridging solutions would be required and functionality duplicated in both C++ and Java by the existing code (and added to by the development of new functionality and the modifications of e.g. data access layers). When the developers estimated the costs associated with using both Tcl and Java in the client (since much code could be reused), and using only one (thus extending the existing code in one language with the functionality of the other), it was concluded that using two different languages in the client would probably be more costly than using either one, due to the same arguments as above. Some generic components, among them non-trivial graphical widgets, would need to be written in two languages.

Building a common data model from existing data models is one of the major challenges of software engineering [1,10], which was apparent from the cost estimations. We cannot claim, on the basis of a single case study, that the “data level” approach will always be as expensive as the “code level” approach, but this reasoning gives at hand that in general, neither approach is cheap, once a minimum of data level integration is decided upon. For the “data level” alternative this requires changes throughout the existing systems and the “code level” alternative requires changes, to adapt to both the new data model and a single set of technologies, languages, and architectural styles.

4.3 Estimated Time to Delivery

The resulting project plans developed in the case study are shown in Figure 4. Although the diagrams presented here are somewhat simplified compared with those developed in the project, they suffice to illustrate some features of this type of project plan:
The definition of a common data model is crucial in both integration approaches, since most other activities are dependent on it. In the case study, the developers were explicit that this activity should not be rushed, and should involve the most experienced users as well as developers.

Management is given a certain amount of freedom by not assigning strict dates to activities. Activities can be prioritized and reordered, and deliveries “spawned off” to meet business demands. More staff can be assigned to certain activities to increase parallelism and throughput. Based on which components would need to be included in a delivery, it is possible to define activities that produce these components; for example, if a delivery with functionality “X” is desired, the activity “Extend with functionality X” or “New functionality X” (for the two alternatives respectively) must be performed as well as all activities on which it is dependent. One strategy could be to aim at delivering a “vertical slice” of the system, incorporating the functionality that is most used first. In this way some users can begin using the new system, thus minimizing the need for maintenance and development of the existing systems (which will soon be retired).

In the “code level” alternative, many activities are of the “transfer functionality” type. In this way, users of the Java system will only see the functionality grow rapidly, but the users of the other systems will experience a period when most of the functionality exists in both the system with which they are familiar and the new system. For the “data level” alternative, the activities are more of the kind “modify the existing systems”. The users would then continue using their familiar system but, when beginning to use the other systems, would have access to more functionality working on the same data. This type of reasoning impacts on long-term planning aspects such as the time at which existing systems can be phased out and retired.
a) Project schedule plan for "data level" alternative:

<table>
<thead>
<tr>
<th>General activities</th>
<th>C++/Motif system</th>
<th>Tcl/C++ system</th>
<th>Java system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define data model</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Develop general functionality</td>
<td></td>
<td>Replace proprietary database</td>
<td></td>
</tr>
<tr>
<td>Modify classes/user interface</td>
<td></td>
<td>Modify classes/user interface</td>
<td></td>
</tr>
<tr>
<td>Extend with functionality X</td>
<td></td>
<td>Extend with functionality Y</td>
<td></td>
</tr>
<tr>
<td>Extend with functionality Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) Project schedule plan for "code level" alternative:

<table>
<thead>
<tr>
<th>General activities</th>
<th>Implement new functionality</th>
<th>Transfer functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define data model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Develop general functionality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New functionality X</td>
<td></td>
<td>Transfer functionality A from Tcl/C++ system</td>
</tr>
<tr>
<td>New functionality Y</td>
<td></td>
<td>Transfer functionality B from Tcl/C++ system</td>
</tr>
<tr>
<td>New functionality Z</td>
<td></td>
<td>Transfer functionality C from C++/Motif system</td>
</tr>
<tr>
<td>New functionality W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: The outlined project plans.

- In the “code level” alternative, it was possible to identify more general components that would require an initial extra amount of effort and
calendar-time but would eventually make the project cheaper and faster. In the “data level” alternative, only few such components were identified.

- Some development of totally new functionality demanded by users was already planned and could not be delayed until the systems integration efforts were completed. However, it was agreed that these activities should be delayed as long as possible – at least until one of the integration alternatives was chosen, and if possible, until the new data model had been defined, and even general components implemented in the case of the “code level” alternative. This was to avoid producing even more source code that would need to be modified during the integration.

4.4 The Decision

When the developers from the two sites had jointly produced these two alternatives and analyzed them, the management was to decide which alternative to choose. It was agreed that the “code level” alternative was considered to be superior to the “data level” alternative from virtually all points of view. The users would experience a more powerful, uniform and homogeneous system. It would also be easier (meaning less costly) to maintain. The analysis had shown that it would include a smaller code base as well as a smaller number of languages, third-party software, and other technologies. The languages and technologies used were more modern, implying that they would be supported by more tools, easier to use and more attractive to potential employees. Not least, the resulting product would be conceptually integrated. Regarding the choice between using Java and Tcl in the client, the management accepted that if the “code level” was decided upon, Tcl would be used since using Tcl implied a significantly smaller effort (due to a larger code base to reuse).

When management considered all this information, they judged the integration to be sufficiently beneficial to motivate the high cost. The benefits included, as we have indicated earlier, increased user efficiency, decreased maintenance costs (in the case of the “code level” alternative), as well as less tangible business advantages such as having an integrated system to offer customers. Also, the evolution scenarios for the existing systems if no integration was performed would be costly; for example, the European organization would probably replace in the near future, the proprietary object-oriented database with a commercial relational database
for maintenance and performance reasons. The cost of implementing the “data level” and “code level” alternatives (when using Tcl in the client) had been estimated to differ insignificantly, and as the organization had to develop it with a limited number of staff, the estimated time to delivery would also be very similar, although the deliveries would be of different kinds due to the different natures of the activities needed for the two alternatives. The relation benefit vs. cost and time to delivery can therefore be visualized as Figure 5 illustrates (the “import/export interface” level was not analyzed, hence the parentheses).

![Figure 5: The estimated cost and time to delivery.](image)

As became clear by now, it was less important to get as much benefit as possible for the cost than to decrease the risk as much as possible. No formal risk analysis was performed at this point, but the risk was judged to be higher for the “code level” alternative, since it involves rewriting code that already exists and works, i.e. risking overrunning schedule and budget and/or decreasing the quality of the product, but also a risk in terms of “commitment required” from the departments of two previously separate organizations, not yet close collaborators. By choosing the “data level” alternative, each system would still be functioning and include more functionality than before, should the integration be discontinued due to e.g. an unacceptable schedule and/or budget situation. This is discernible in the project plans of Figure 4. Management doubted that the cost of the two alternatives would really be similar; they intuitively assumed that the higher benefit, the more effort was required (cost and time), as was sketched in Figure 2. Still, they were explicit in that the risk was the decisive factor and not cost, when choosing the “data level” alternative.
5. Related Work

There are suggestions that project management during ordinary software development has much to gain from being “architecture-centric” [21]. We have shown some ways of pursuing the architecture-centric approach during integration also. The rest of this section will focus on two related aspects of this, the literature relating to integration approaches, and methods and analysis techniques based on architectural descriptions.

Of the four integration approaches we have discussed, Enterprise Application Integration (EAI) seems to be the most documented [9,12,15,19,22]. This approach concerns in-house integration of the systems an enterprise uses rather than produces. Johnson [15] uses an architectural approach to analyze the integration of enterprise software systems. In spite of the difficulty of accurately describing the architecture of this type of system because the available documentation is inadequate, architectural analysis can be successfully applied to the design of enterprise systems integration. Johnson has also examined the limitations of architectural descriptions which one must be aware of, limitations that were also experienced in the case study.

None of the architectural methodologies available were completely feasible for the task. The Architecture Trade-off Analysis Method (ATAM) [6] and the Software Architecture Analysis Method (SAAM) [2,6] are based on stakeholder-generated scenarios. The ATAM requires business drivers and quality attributes to be specified in advance and more detailed architectural descriptions to be available. In the case study, all of this was done in a more iterative manner. Also, with limited resources, it would be impossible to evaluate and compare several alternatives, it being too time-consuming to investigate all combinations of quality attributes for all working alternatives. While both SAAM and ATAM use scenarios to evaluate maintainability, we used another, if less accurate measurement method, comparing the number of lines of code, third-party software, languages, and technologies used, assuming that the lower the number, the easier the maintenance. The Active Reviews for Intermediate Designs method (ARID) [6] builds on Active Design Reviews (ADR) and incorporates the idea of scenarios from SAAM and ATAM. It is intended for evaluating partial architectural descriptions, exactly that which was available during the project work. However, it is intended as a type of formal review involving more stakeholders and this was not possible because the project schedule was already fixed, and too tight for an ARID exercise. All of these methodologies analyze functionality (which was relatively trivial in the case study as the integrated system would
have the functionality of the three systems combined) and quality attributes such as performance and security (which are of course important for the product of the case study, but considered to be similar to the existing systems) – but none addresses cost, time to delivery, or risk, which were considered more important. The project therefore relied more on the analysts’ experience and intuition in analyzing functionality and quality attributes (because of the project’s limited resources), and cost, time to delivery, and risk (because there are no available lightweight methodologies for analyzing these properties from architecture sketches).

6. Conclusions

We have shown the central role of software architecture in a case study concerning the integration of three software systems after a company merger. Some important lessons we learned from this case study can be formulated as follows:

- There are at least four approaches available to a software integrator: Enterprise Application Integration (EAI), interoperability, data level integration, and source code integration. The choice between these is typically based on business or organizational considerations rather than technical.

- When the architectural descriptions of existing systems are not easily comparable, the first task is to construct similar architectural descriptions of these. The components of the existing systems can then be rearranged in different ways to form different alternatives. The working alternatives can be briefly analyzed, largely on the basis of known properties of architectural patterns of the existing systems.

- The functional requirements of an integrated system are typically a combination of the functionality of the existing systems, and are relatively easy to assess as compared with other quality attributes.

- The effort required to implement each component of the new system can be estimated in terms of how much can be reused from the existing systems and how much must be rewritten. The total cost of the system is easily calculated from these figures.

- According to the estimations performed in the case study, source code level integration is not necessarily more expensive than data level integration.
• Architectural analysis, as it was carried out in the project, fails to capture all business aspects important for decisions. All the information needed to produce a project schedule is not present in an architectural description. The risk associated with the alternatives was identified as the most important and least analyzed decision criteria.

There are a number of concerns that must be addressed during integration planning as well as during software activities in general. These include the process and time perspective (e.g. will the integration be carried out incrementally, enabling stepwise delivery and retirement of the existing systems?), the organizational issues (e.g. who are the stakeholders?), the cost and effort requirements (e.g. are only minimal additional efforts allowed?), etc. We have shown how a system’s architecture can be used as a starting and central point for a systematic analysis of several features. To what extent can such concerns be addressed by architectural analysis? Perhaps the focus on the architecture, basically a technical artifact poses a risk to these other concerns? We have presented means of estimating cost and time of implementation based on architectural descriptions, including outlining project schedules. We have also shown that only the parts of such project schedules involving implementation of source code can be produced from the architectural descriptions, activities such as design or analysis must be added from other sources. We also showed that the risk of choosing one alternative or the other was not considered. We therefore propose that risk analysis be included in architectural analysis to make it more explicit (or the opposite, that architectural analysis be used in project risk analysis). This would make it possible to treat risk together with other quality properties and make a conscious trade-off between them. Research in this area will presumably need to incorporate an organizational development and production process model – which would also provide a better basis for time and cost estimation.

7. References


Paper II

This paper is a reprint of:

The open-ended interview questions used to collect data are reprinted in Appendix B.
(Not in thesis format yet.)
Manuscript Number: INFSOF-D-06-00012R1

Title: Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection

Article Type: Research paper

Section/Category:

Keywords:

Corresponding Author: Mr Rikard Land, M.Sc.

Corresponding Author's Institution: Mälardalen University

First Author: Rikard Land, M.Sc.

Order of Authors: Rikard Land, M.Sc.; Ivica Crnkovic, Ph.D.

Manuscript Region of Origin:
Software Systems In-House Integration:  
Architecture, Process Practices and Strategy Selection

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.

For an organization that has identified a functional overlap and a need for integration, two main questions appear: what will be the final result, and how it can be achieved? That is, how do we devise a vision for a future integrated system, and how do we utilize our existing systems optimally in order to reach this vision within reasonable time, using reasonable resources?

The main difficulty in this situation is not to invent or develop new software, but to take advantage of the existing proven implementations as well as of the collected experience from
developing, evolving, and using the existing systems. We have labeled this general approach *in-house integration*, which might be solved in a number of fairly different ways, from retiring existing systems (but reuse experience), through integration of components of the original systems to performing a very tight merge of pieces of code. As the initial decisions will have far-reaching consequences, it is essential to analyze important factors in enough depth prior to making the decision. In this paper we describe a number of such factors we have observed and derived from the case studies, and we also present some observations on the consequences of ill-founded decisions.

1.1 Research Relevance and Scope

There is no established theory or methods that specifically address this situation. As there is arguably much experience in industry, research on this should start by collecting this experience. Being the first study of its kind, this study is qualitative in nature, and our research method (further described in section 2) has been the multiple case study. The industrial cases all have all a considerable history of usage and evolution (often twenty years or more). They range from moderately small (a few developers) to large (hundreds of developers). The relevance of the cases for the research problem is thus high, and the number of cases ensures some level of external validity.

We have deliberately chosen to not study enterprise information systems, because there are research and textbooks available on Enterprise Application Integration (EAI) already, and there exist commercial solutions for this as well as a large number of consulting companies [10,35,39,63,79]. We have instead searched for cases in other domains, and studied how they have carried out their software systems integration, and how well they consider themselves to have succeeded. The domains of the systems include safety-critical systems (i.e. including hardware), physics simulations, software platforms, human-machine interfaces, and data management systems. These domains are not so suited for EAI interconnectivity approaches, which would imply extra overhead of adapters, wrappers, etc. Also, one of the main goals of in-house integration – to reduce the maintenance costs – would not be met if the amount of source code is extended rather than reduced.

1.2 The Big Picture

The total process is typically divided into a *vision process* and an *implementation process*. Even if this is not always done explicitly, there is a clear difference between the purpose of each, the participants of each, and the activities belonging to each [58]. The starting point is a decision that integration is needed or desired, and the outcome of the vision process is a high-level description what the future system will look like, both in terms of features (requirements) and design (architectural description). It would also include an implementation plan, including resources, schedule, deliverables, etc. The implementation process then consists of executing the plan. Different stakeholders are involved at different stages; during the strategic vision process, managers and architects are heavily involved, while developers become involved mainly during implementation.

These processes could be as sequential – first define a vision, then implement it. If the decision is drastic, involving e.g. discontinuing one system immediately, there would typically be no need to revisit it after implementation has made some progress. Otherwise, the vision will likely be revisited and refined as implementation proceeds, meaning that these two processes will be carried out somewhat iteratively, each affecting the other. This is similar to many other software process models: for new development there are sequential processes as well as iterative processes. Which is most feasible in any given case depends on e.g. how
often the requirements might change, how well the developing organization knows the domain, and the type of system. As we are here dealing with existing systems, it becomes even more important to make an explicit decision on what is wanted before starting implementing it.

The vision process will end in a high-level description of what the new system should look like. There are of course many possible designs of the system, but at the highest level there seem to be a few, easily understood strategies, characterized by the parts of the existing systems that are reused. It is possible to formulate four fundamental strategies: No Integration, Start from Scratch, Choose One, and Merge. Their primary use is as a vehicle for discussion, and we can expect that in most cases, the solution chosen could be characterized as somewhere in between. By formulating the strategies in their pure form, any particular solution in reality can be characterized in terms of these extreme ends of the solution space.

- **Start from Scratch** Start the development of a new system, aimed to replace the existing systems, and plan for discontinuing the existing systems. In most cases (parts of) requirements and architecture of the existing systems will be carried over to the new system. This strategy can be implemented by acquiring a commercial or open source solution, or building the new system in-house.

- **Choose One** Evaluate the existing systems and choose the one that is most satisfactory, officially discontinue development of all others and continue development of the selected system. It may be necessary to evolve the chosen system before it can fully replace the other systems.

- **Merge** Take parts from the existing systems and integrate them to form a new system that has the strengths of both and the weaknesses of none. This is, of course, an idealized strategy and as it will turn out the most complicated and broad strategy of the model. To aid the discussion of the paper, we present two types of Merge, labeled Rapid and Evolutionary, only distinguished by their associated time scale. By introducing them, some events in the cases and some conclusions are more easily explained, although there is no strict borderline between them.
  - **Rapid Merge** With “rapid” we mean that the existing components can be rearranged with little effort, i.e. basically without modification or development of adapters. How to evaluate and select components is the responsibility of the architect.
  - **Evolutionary Merge** Continue development of all existing systems towards a state in which architecture and most components are identical or compatible, in order to allow for Rapid Merge sometime in the future.

- **No Integration** This strategy is mentioned solely to be complete. No integration means that the existing software systems are controlled independently, which clearly will not result in an integrated or common system.

Some notes on terminology before continuing, to distinguish between our uses of integration and merge: The term in-house integration is used to denote the general approach of providing a single implementation, reusing whatever could be reused. Integration may or may not be implemented by the Merge strategy. With a new generation of a system we intend a system developed from scratch, i.e. without reusing code. We are not making any fine distinction
between the terms *maintenance* and *evolution*. The term *architectural compatibility* is used to describe the relationship between two systems (how “similar” are they, in some sense), not between two components (which would be how “composable” are they [67]).

### 1.3 The Present Paper

In this paper we focus on the vision process. To do this we have two types of data from the cases: first, direct observations from the vision process, and second, observations on the implementation process (and the outcome, although in some cases integration is still in progress). This second type of data is interesting as narratives in themselves, but is also used to suggest elements that should be included in the vision process.

Details regarding the research design are found in section 2, including brief narrative descriptions of the events in the cases. Section 3 contains direct observations from the vision process, which addresses the first goal: we report process practices found in the cases and introduce the criteria for selecting a strategy. These criteria are elaborated in subsequent sections, thus addressing the second goal: architectural compatibility in section 4, retireability in section 5, implementation process in section 6, and resources, synchronization, and backward compatibility in section 7. Section 8 re-analyzes the cases based on the criteria presented, and synthesizes these criteria into a suggested high-level check-list procedure. Section 9 relates our work to existing publications, and section 10 summarizes and concludes the paper.

Parts of the present paper have previously been published at various conferences [48,50,52-55,58]. The present paper relates the findings previously reported separately to each other, performs a deeper analysis and provides an overall list of important factors, suggested solutions and practices.

### 2 Research Method

The multiple case study [95] consists of nine cases from six organizations. The cases were found through personal contacts, which led to further contacts, etc. until we had studied the nine cases and proceeded to analysis. Our main data source has been interviews, although in some cases the interviewees offered documents of different kinds (system documentation as well as project documentation), which in our opinion was mostly useful only to confirm the interviewees' narratives. In one case one of the authors (R.L.) participated as an active member during two periods (some three to four months on each occasion, with two years in between).

The desired interviewees were persons 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. There are guidelines on how to carry out interviews in order to e.g. not asking leading questions [77], which we have strived to follow. The questions were open-ended [82], focused around architecture and processes, and the copied out interview notes were sent back to the interviewees for feedback and approval. The interviewees were asked to describe their experiences in their own words, as opposed to answering questions; however we used a set of open-ended questions, to ensure we got information about the history of the systems, problems, solutions, and more. The questions are reprinted in the Appendix. Due to space limitations the answers are not reprinted, but can be found in a technical report together with further details regarding the research design [57]. Our analysis protocol was as follows: the responses were read through many times, each time with a particular question in mind (e.g. searching for anything related to “Small Evaluation Group” or “Stepwise deliveries”). This reading was guided by preliminary propositions made; these were the results of previous experiences, repeated discussions of the responses during the data collection phase. In the end, we ensured that all responses had been read through at least once from every point of view, by at least one researcher.

The research can be considered to be grounded theory [86] in the sense that we collected data to build models for previously un-researched questions, as contrasted to validating a pre-defined hypothesis. The ideas were refined already during data collection, and the questions were more directed during the last interviews. Strictly, this gives no external validity in the traditional sense – the data fits the model, because the model is built from the data, and it has been argued that the validation that can be achieved by this method is a proper understanding, which can only be judged by others [64,86]. After this multiple case study, we have initiated a validation phase, in which we by means of a questionnaire are validating and quantifying the results presented in the present paper. So far, a few organizations have been studied and the preliminary results are in line with the present paper [59,60].

We have deliberately avoided labeling the outcome of the cases as being good or bad, as the criteria as to how to do this are not at all obvious and are practically difficult to determine. Problems in answering this question include: how many years need to pass before all effects of the integration are known? How can the quality of the resulting systems be evaluated, if at all? (Some quality metrics could be used such as defect detection rates, number of user reports, measures of complexity [25,31,70], surveys of user satisfaction.) Or is the competitiveness and financial situation of the company a certain number of years a more interesting measure? When should return on investment be evaluated, and how can we be sure that this can be attributed to the integration and nothing else? An inherent limitation of case studies is that it is impossible to know what the result of some other choice would have been. All value statements therefore come from the interviewees themselves, based on their perception of e.g. whether time and money was gained or wasted.

2.1 Limitations

As the cases represent a wide variety and size of organizations, domains, and systems, a natural question is to what extent the observations indeed are general, or only apply to similar organizations, domains, or systems. It appears that the problems and solutions are general (for example that a smaller team may be managed more informally than a large, or that a safety-critical system would require a stricter process than an entertainment product [9]). This means that the diversity among the cases is a strength which increases the external validity of the findings. On the other hand, it could also be argued that we could not reliably find any systematic differences between the cases, as we have too few of each type and size of organization, domain, and system. It might also be argued that the way cases were selected
(mainly through personal academic contacts) systematically would exclude some type of cases (e.g. organizations with little collaboration with the academic world).

The cases include only western cultures. It is possible that some of our observations are of less importance in other parts of the world, and that a study including organizations in other cultures would come up with additional observations. For example, the small evaluation group process practice (section 3.2) to some extent assumes a method of making decisions involving many different points of view; in a more authoritative culture (i.e. with larger “power distance” [34]), a small evaluation group would perhaps not be an appropriate way of making a decision. Matters become even more complex when considering that international mergers and acquisitions will involve two different cultures, which will need to find common denominators to be able to work together.

We have approached the problem and formulated questions in terms of the systems (at a high level) and about processes. Other viewpoints could focus more on people and psychology, and study how to make processes and practices actually work [80].

2.2 The Cases

The cases come from different types and sizes of organizations operating in different business domains. The case studies have included global and distributed organizations and collaborations, including (in alphabetical order): ABB, Bofors, Bombardier, Ericsson, SAAB, Westinghouse, and the Swedish Defence Research Agency. Several cases involve intercontinental collaborations, while organizations B and E are national (but involve several sites). The size of the maintenance and development organizations range from a few people to several hundred people, and the system qualities required are very different depending on the system domain. What the cases have in common though is that the systems have a significant history of development and maintenance.

The cases are summarized in Table 1. They are labeled A, B, etc. in line with previous publications [48,50,52-55,57,58]. Cases E1, E2, F1, F2, and F3 occurred within the same organizations (E and F). Throughout the paper, we will refer to the cases and often point into specific sources of data. For these data sources, the acronyms used are I_X for interviews, D_X for documents, and P_X for participation, where X is the case name (as e.g. in I_A, the interview of case A), plus an optional lower case letter when several sources exist for a case (as e.g. for interview I_Da, one of the interviews for case D). I_X:n refers to the answer to question n in interview I_X. All data sources can be found in the technical report [57]. For direct quotes, quotation marks are used (“”).

Some cases have successfully performed some integration, others are underway. All cases reported both successes and relative failures, which are all taken into account in the present paper.

The rest of this section presents details for all of the nine cases. One of the cases (case F2) is described in depth, followed by more summarized descriptions of the others. The motivation for selecting case F2 for the in-depth description is that it illustrates many of the concepts brought forward in the paper. It is also the case with the largest number of interviews made (six), and one of the authors (R.L.) has worked within the company (in case F1) and gathered information not formalized through interview notes. We have chosen to keep the case labels consistent with previous publications, which explains why the cases are not labeled according to the order in which they are presented.
2.2.1 Case F2: Off-line Physics Simulation

Organization F 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, physics computer simulations are conducted. Central for many simulations made is a 3D simulator consisting of several hundreds of thousands lines of code (LOC) (I_F2e:1, I_F2f:1). Case F2 concerns two simulation systems consisting of several programs run in a sequence, ending with the 3D simulator (I_F2a:1, I_F2b:1). The pipe-and-filter architecture [12] and the role of each program is the same for both existing systems, and all communication between the programs is in the form of input/output files of certain formats (I_F2a:1,9, I_F2b:7, I_F2c:10,11, I_F2d:8, I_F2e:5, I_F2f:8). See Figure 1.

The 3D simulator contains several modules modeling different aspects of the physics involved. One of these modules, which we can call PX (for “Physics X”), needs as input a large set of input data, which is prepared by a 2D simulator. In order to help the user preparing data for the 2D simulator, there is a “pre-processor”, and to prepare the PX data for the 3D simulator, a “post-processor” is run; these programs are not simple file format translators but involve some physics simulations as well (I_F2a:1, I_F2b:1).

It was realized that there was a significant overlap in functionality between the two simulation systems present within the company after the merger. It was not considered possible to just discontinue either of them and use the other throughout the company for various reasons. In System 1 (the US system), a more sophisticated 2D simulation methodology was desired, a methodology already implemented in the System 2 (the Swedish system) (I_F2a:3). In System 2 system on the other hand, fundamental problems with their mathematical model had also been experienced; it was e.g. desired to separate different kinds of physics more to make it more flexible for different types of simulations (I_F2a:3). In addition, there are two kinds of simulations made for different customers (here we can call them simulations of type A and B), one of which is the common type among System 1’s customers, the other common among System 2’s customers (I_F2a:1, I_F2c:10). All this taken together led to the formation of a common project with the aim of creating a common, improved simulation system (I_F2c:3). The pre-processor, post-processor, and parts of the 3D simulators are now common, but integration of the other parts are either underway or only planned. There are thus still two distinct systems. The current state and future plans for each component are:

- **Pre-processor.** The pre-processor has been completely rewritten in a new language considered more modern, i.e. *Start from Scratch* (I_F2b:1,7). Based on experience from previous systems, it provides similar functionality but with more flexibility than the previous pre-processors (I_F2b:7). It is however considered unnecessarily complex because two different 2D simulators currently are supported (I_F2b:7,9).

- **2D Simulator.** Although the 2D simulators are branched from a common ancestor, they are no longer very similar (I_F2a:1, I_F2b:7, I_F2d:7,8). By evolving the simulator of system 1, a new 2D simulator is being developed which will replace the existing 2D simulators, i.e. *Choose One* (I_F2a:9, I_F2b:7, I_F2d:7,8). It will reuse a calculation methodology from System 2 (I_F2a:3, I_F2c:9). Currently both existing 2D simulators are supported by both the pre- and post-processor (I_F2b:7,9, I_F2d:7).

- **Post-processor.** It was decided that System 2’s post-processor, with three layers written in different languages, would be the starting point, based on engineering judgments, i.e. *Choose One* (I_F2a:7, I_F2c:7, I_F2d:6). This led to large problems as the fundamental assumptions turned out to not hold; in the end virtually all of it was rewritten and
restructured, i.e. Start from Scratch although still with the same layers in the same languages (If2a:9, If2c:7,9, If2d:6,7, If2e:7).

- **3D simulator.** The plan for the (far) future is that the complete 3D simulator should be common, i.e. Evolutionary Merge (If2a:3, If2c:3, If2d:3). “X” physics is today handled by a new, commonly developed module that is used in both the Swedish and US 3D simulators (If2e:7). It has a new design, but there are similarities with the previous modules (If2d:7). In order to achieve this, new data structures and interfaces used internally have been defined and implemented from scratch (If2c:7, If2d:6,7); common error handling routines were also created from scratch (If2c:7); these packages should probably be considered part of the infrastructure rather than a component. All this was done by considering what would technically be the best solution, not how it was done in the existing 3D simulators (If2d:7,8, If2f:6). This meant that the existing 3D simulators had to undergo modifications in order to accommodate the new components, but they are now more similar and further integration and reuse will arguably become easier (If2e:7).

To create a common system, it was considered possible to discontinue the first three parts, as long as there is a satisfactory replacement, although the simulators need to be validated which makes time to release longer (If1c:6, If1f:6). Figure 2 shows the current states of the systems. It should be noted that although there are two 3D simulators, some internal parts are common, as described by Figure 3.

### 2.2.2 Other Cases
This section presents the most relevant observations in each of the remaining cases.

**Case A.** Organization: Newly merged international company. System domain: Safety-critical systems with embedded software. To avoid duplicated development and maintenance efforts, it was decided that a single human-machine interface (HMI) platform should be used throughout the company, instead of the previously independently developed HMIs for their large hardware products (IA:1,2,3). One of the development sites was considered strongest in developing HMIs and was assigned the task of consolidating the existing HMIs within the company (IA:2). New technology (operating system, component model, development tools, etc.) and (partly) new requirements led to the choice of developing the next generation HMI platform without reusing any implementations, but reusing the available experience about both requirements and design choices (IA:5,6,7). This also included reuse of what the interviewee calls “anti-design decisions”, i.e. learning from what was not so good with previous HMIs (IA:7). The most important influence, apart from their previous experience in-house, was one of the other existing HMIs which was very configurable, meaning it was possible to customize the user interface with different user interface requirements, and also to gather data from different sources (IA:3,5). These two systems had very different underlying platforms: one was based on open source platforms and the other on commercial solutions (IA:1,2,8) which – in the context of this company – excluded the possibility of a Merge in practice. As resource constraints were not a major influence, the decisive factor when choosing between the remaining strategies was the new consolidated set of requirements, especially larger configurability of the system, and the availability of new technology (IA:3,5). Therefore, Start from Scratch was desired by the architects over Choose One and thus selected (IA:7,8).

**Case B.** Organization: National corporation with many daughter companies. System domain: Administration of stock keeping. One loosely integrated administrative system had been customized and installed at a number of daughter companies (Ib:1). In one other daughter
company, a tightly integrated system had already been built, but for the future, it should be merged with the loosely integrated system (IB:3). The large system had rich functionality and high quality, and was installed and used in many daughter companies. Discontinuing it would be a waste of invested resources and was not even considered as an option, i.e. Start from Scratch was excluded. The smaller system was built on the tight integration paradigm, while the large system was built as a loose integration of many subsystems (IB:1,6,7,13). The difference in approach made Merging the systems infeasible, therefore the Choose One strategy was chosen (IB:7). The total integrated system was stripped down piece by piece, and functionality rebuilt within the technological framework of the loosely integrated system (IB:3,7). Many design ideas were however reused from the totally integrated system (IB:7).

**Case C.** Organization: Newly merged international company. System domain: Safety-critical systems with embedded software. The systems and development staff of case C is the largest among the cases: several MLOC and hundreds of developers (ICb:1,9). The systems’ high-level structure were similar (ICa:7, ICb:1), but there were differences as well: some technology choices, framework support mechanisms such as failover, the support for different (natural) languages, and error handling were designed differently, and there was a fundamental difference between providing an object model or being functionally oriented (ICa:1,6,7). At the time of the company merger, new generations of these two systems were being developed and both were nearing release (ICa:1, ICb:1). Senior management first demanded reuse of one system’s HMI and the other’s underlying software within six months, i.e. Rapid Merge, which the architects considered unrealistic (ICa:6,8, ICb:6). The strict separation of software parts implied by being embedded in different hardware parts could make Evolutionary Merge possible, requiring two years. Merge was, according to the architects, the wrong way to go, and a better option would be to Choose One (ICa:6, ICb:6). However, management seemed unable to make a decision, and as time passed both systems were independently released and deployed at customers. Eventually management allowed that either (but not both) could be discontinued, allowing for the Choose One strategy which was implemented (ICa:6). The delay caused an estimated loss of one year of development effort of a team of several hundred developers, confusion to the customers who did not know which of the two products to choose, and required addition effort in migrating the customers of the retired system to the chosen one (ICa:6, ICb:6). One of the interviewees points out that although the process seems less than satisfactory it is difficult to say whether other approaches would have been more successful (ICa:12). Once this decision was made, reusing some components of the discontinued system into the other became easier (ICb:7). It took some twenty person-years to transfer one of the major components, but these components represented so many person-years that this effort was considered modest compared to rewrite, although (with the words of one interviewee) “the solutions were not the technically most elegant” (ICb:7).

**Case D.** Organization: Newly merged international company. System domain: Off-line management of power distribution systems. After the company merger, the two previously competing systems have been continued as separate tracks offered to customers; some progress has been made in identifying common parts that can be used in both systems, in order to eventually arrive at a single system (IDa:1,3, IDb:5,6). The two systems, both consisting of a client HMI and a server, have a common ancestry, but have evolved independently for 20 years (IDa:1). System 1’s HMI built on more aged technologies, and around the time of the merger the customers of this system considered the HMI to be outdated; it was therefore decided that it should be replaced by System 2’s more modern user interface, thus Choose One (IDa:1, IDb:3). In System 1, two major improvements were made: System 2’s HMI was reused, and a commercial GIS (Geographic Information System) tool was acquired (instead of System 2’s data engineering tool) (IDa:1, IDb:3,8). Five years before
the merger System 2’s HMI was significantly modernized which made reusing it possible thanks to the new component-based architecture (I Da:1, IDb:3,7,8). Its component framework made it possible to transfer some functionality from System 1’s HMI by adding and modifying components (I Db:7). Also contributing to the possibilities for using System 2’s HMI with System 1’s server was the similarities between the systems, both from the users’ point of view (IDb:6) and the high-level architecture, client-server (I Da:7, IDb:7,8); these similarities were partly due to a common ancestry some twenty years earlier (I Da:1). This made it relatively easy to modify the server of System 1 in the same way as the server of System 2 had been modified five years ago when the modern HMI was developed (IDb:8). The servers themselves are still separate; they both implement the same industry standards and the plans are to perform an Evolutionary Merge but there are yet no concrete plans (I Da:1, IDb:6).

Case E1. Organization: Cooperation defense research institute and industry. System domain: Off-line physics simulation. Several existing simulation models were to be integrated into one. The goal was not only to integrate several existing systems, but also to add another, higher level of functionality (I E1:1,3). Retiring the existing systems was possible since all parties would benefit from the new system (I E1:1). There were a number of existing simulation models, implemented in FORTRAN and SIMULA, which would make reuse into an integrated system difficult (I E1:6). Also, the new system would require a new level of system complexity for which at least FORTRAN was considered insufficient; for the new system Ada was chosen and a whole new architecture was implemented using a number of Ada-specific constructs (I E1:6,7). Ada would also bring a number of additional benefits such as reusable code, robustness, commonality within the organization (I E1:6,7). Many ideas were reused however, and transforming some existing SIMULA code to Ada was quite easy (I E1:7). Thus Start from Scratch strategy was chosen (I E1:6).

Case E2. Organization: Different parts of Swedish defense. System domain: Off-line physics simulation. A certain functional overlap among three simulation systems was identified (I E2:1, DE2a). The possibility of retiring any, but not all, of these systems was explicitly left open, partly because of limited resources and partly because (some of) the functionality was available in the others (DE2a, I E2:13). System 1 and System 2 were somewhat compatible, but due to very limited resources the only integration has been System 1 using the graphical user interface of System 2 (I E2:6). The two systems use the same language (Ada) and the integration was very loose (I E2:7). Nevertheless, reusing System 2’s user interface required more effort than expected, due to differences in input data formats, log files, and the internal model (I E2:7). We thus have some reuse but no Merge, as there are no resources and no concrete plans for integrating these two systems into one. Although not directly replaced by the others, System 3 has in practice been retired (I E2:6,13) and we consider this case to be a Choose One strategy (actually Choose Two out of three).

Case F1. Organization: Newly merged international company. System domain: Managing off-line physics simulations. After the company merger, there has been a need to improve and consolidate management and support for certain physics simulations, focused around the major 3D simulators used (those described in case F2), but also including a range of user interfaces, data management mechanisms, and automation tools for different simulation programs (IF1a:1, IF1b:1, IF1c:1,2, DF1a, PF1a, PF1b). An ambitious project was launched with the goal of evaluating the existing systems. There were differences in architectures, programming languages, technologies, and data models (IF1a:6, IF1b:6, IF1c:6,7,9, DF1a, PF1a, PF1b). It was not considered possible to discontinue development on the existing systems before a full replacement was available (IF1c:6, DF1a, PF1a, PF1b). Two possibilities were outlined: a tight merge with a result somewhere between Merge and Choose One, and a loose integration
where the systems would share data in a common database in an *Evolutionary Merge* manner; the loose integration alternative was eventually chosen (I_F1a:3, I_F1c:3, P_F1a, D_F1a). The many differences indicated a very long development time, and it appears as this solution was perceived as a compromise by the people involved, and was followed by no concrete activities to implement the decision (I_F1c:6, P_F1a, P_F1b). Later, there have been numerous other small-scale attempts to identify a proper integration strategy, but the limited resources and other local priorities in practice have resulted in no progress towards a common system (I_F1a:3, I_F1b:9,11, I_F1c:6, D_F1a, P_F1a, P_F1b). Currently, at least some stakeholders favor *Choose One*, but the scope is unclear (discussions tend to include the whole software environment at the departments) and integration activities still have a low priority (I_F1a:1,6, I_F1b:6,9, I_F1c:1, P_F1b). Some participants have seriously begun to question the value of integration altogether (I_F1b:3,9, I_F1c:6,9) and the result so far, after four years, has been *No Integration*.

**Case F3.** *Organization: Newly merged international company. System domain: Software issue reporting.* Three different software systems for tracking software issues (errors, requests for new functionality etc.) were used at three different sites within the company, two developed in-house and one being a ten-year old version of a commercial system (I_F3:1). The two systems developed in-house were somewhat compatible (I_F3:1). All involved saw a value in a common system supporting the best processes within the company, and apart from the fact that a transition to such a common system would mean some disruption at each site, independently of whether the common system would be completely new or a major evolution of the current system used there was no reluctance to the change (I_F3:3,10,11). Being a mature domain, outside of the company’s core business, it was eventually decided that the best practices represented by the existing systems should be reused, and a configurable commercial system was to be acquired (i.e. the organization chose to *Start from Scratch* by acquiring a commercial solution) and customized to support these (I_F3:6).

### 3 Vision Process

The starting point is often an initial vision from senior management (I_Ca:6, I_Cb:6, I_Db:3,5,6, I_F2c:3). The goal is to rationalize the activities related to the products by e.g. rationalizing maintenance, reducing data overlap, and avoiding duplicated processes (I_A:2,3, I_B:1, I_Ca:6, I_Cb:6, I_Db:3,5,6, P_F1a, P_F1b, D_F1b, I_F2d:3).

In the rest of this section we will introduce the criteria for selecting a strategy (section 3.1, to be elaborated throughout the rest of the paper), and in section 3.2 discuss the process practices found in the cases.

#### 3.1 The Goal of the Vision Process

The expected outcome of the vision process is the selection of one of the strategies presented earlier: *Start from Scratch, Choose One, Merge* (or possibly *No Integration*), and outlines of the future system and the implementation process. Selecting a strategy is in any real-life situation naturally influenced by many factors of many different kinds, such as: the current state of the existing systems, both technically and management aspects; the level of satisfaction with existing systems among users, customers, and the development organization; the completeness or scope of the existing systems with respect to some desirable set of features; available development resources; desired time to market, etc.

A reasonable starting point for a systematic approach would be to early focus on questions and issues that could rule out some strategies. Based on the cases, we have found two such main concerns that, when properly addressed, can help excluding strategies:
- **Architectural compatibility.** Depending on how similar the existing systems are in certain respects, integration will be more or less difficult. The strategies Rapid Merge will not be possible if the systems are not very similar, and if they are even less compatible, Evolutionary Merge may also be excluded. Start from Scratch and Choose One are essentially not affected by the compatibility of the existing systems. More exactly what types of similarities are the most important, and how to evaluate the existing systems, is not obvious though, and will be elaborated in depth in section 4.

- **Retireability.** Stakeholders may consider retiring a system unfeasible for various reasons, such as market considerations, user satisfaction, or potentially the loss of essential functionality. If the final statement is that no existing systems can be retired, the strategies Start from Scratch and Choose One can be excluded. However, retireability is not a static property in the same sense as the compatibility of the existing systems, but is negotiable. We acknowledge the difficulties of retiring and replacing a system that is in use, but we suggest that the impact on the strategy selection must be understood by the various stakeholders that might have an opinion. Although important, the question of retireability has not been the focus of our research and will in the present paper not be elaborated to the same extent as compatibility.

We do not suggest that compatibility and retireability are more important than other considerations that also influence the choice of strategy. Nor do we suggest a strictly sequential process where compatibility and retireability are evaluated first, and other issues are only considered after the initial exclusion of strategies. In practice all considerations will be discussed more or less simultaneously.

Table 2 summarizes the exclusion of strategies (black denotes exclusion).

It is obvious that it is highly undesirable to arrive at the conclusion that the systems are not compatible, but that none of the existing systems can be retired. The remaining strategy No Integration is – according to our experience from the cases – usually considered the worst option, bringing long-term problems with double maintenance and evolution costs and users having to learn and use several similar systems for similar tasks. However, it should be mentioned that some interviewees proposed the opinion of not integrating at all. “Why integrate at all?” (Ic3:7) is indeed a valid question, which will arise if a decision is not accompanied with priority and enough resources (IF1b:3, IF1c:6,9,11, F1a). Sometimes it might simply not be worth the effort to integrate – will the future savings through rationalization be larger than the integration efforts (IF1c:9, IF2d:3)? In case E2 there were very few resources available, which led to a very modest vision, in practice meaning no integration (IE2:6).

We further describe and analyze the selection criteria as follows: architectural compatibility in depth (section 4), and retireability more briefly (section 5), followed by other influences: the strategies’ implication on the implementation process (section 6), and resources, synchronization, and backward compatibility (section 7). But before that, we present beneficial practices found for the vision process.

### 3.2 Suggested Vision Process Practices

This section describes vision process practices repeatedly suggested among the cases. These are issues emphasized by the interviewees in the cases, sometimes based on positive experiences, but sometimes based on negative experiences, i.e. that they in retrospect ascribe some negative effects to the lack of these practices. We have formulated the lessons learned as statements describing what should be done, not necessarily how things were done in the
cases. For each practice identified, we have argued whether it is inherent and unique to the in-house integration context or is known from other software activities. We have identified two external causes for some of the practices: the distributed organizations and the (typically) long time scale required. These are typical characteristics of the in-house integration context, but are not unique to it – however, not being unique does not make these practices less important for integration.

Two practices were found that seem unique to the context of in-house integration, because they explicitly assume there are two (or more) overlapping systems, and two previously separate groups of people that now need to cooperate:

**Proposition I: Small evaluation group.** After senior management has identified some potential benefits with integration, a small group of experts should be assigned to evaluate the existing systems from many points of view and describe alternative high-level strategies for the integration.

In cases C and F1 a small group evaluated the existing systems with the specific goal to identify how integration should or could be carried out, at the technical level (I_C6;6, I_C6;6, F1a;6, F1b;6, D_E1a). Case F1 involved not only developers but also users and managers at different stages with different roles; the users graded different features of the existing systems and the managers were responsible for making the final decision (P_F1a; D_F1a) [55]. It should be pointed out that during other types of software activities, such as new development and evolution, one should involve all relevant stakeholders [45]. What makes in-house integration unique in this respect is that there are two of each kind of stakeholder: users of system A, users of system B, developers of system A, developers of system B, etc. It is therefore crucial to involve both sides, as no single individual has overview of all systems (both cases C and F1 concern newly merged companies). Also, everyone involved is likely to be biased and there is a clear risk that the participants “defend” their own system (I_C6;6), there must be an open mind for other solutions than “ours” (I_F3;11). In the cases it appears that there has indeed been a good working climate with a “good will” from everyone (I_C6;6, P_F1a). In both cases this was considered a good scheme; in case C the architects immediately saw that there were no major technical advantages of either system, and wanted to immediately discontinue one of the two systems, indifferent which, rather than trying to merge the systems (I_C6;6). The late decision (indeed, to discontinue one of the systems) was due to other reasons (see “timely decisions” below). A similar scheme was used in case E2, where an external investigation was made, however with less technical expertise (I_E2;6, D_E2a).

**Proposition II: Collect experience from existing systems.** All experience of the existing systems, in terms of e.g. user satisfaction and ease of maintenance must be collected in order to be able to describe the envisioned system properly (I_A;6, P_F1a; D_F1a, I_F2c;6, I_F2d;6, I_F3;11).

Ideally, one would like to define the new system as consisting of the best parts of the existing systems; however, this is in practice not as simple as it first may seem. The requirements on the future system are clearly dependent on the experience of the previous systems, and can be stated in terms of existing systems (I_X;6, P_F1a; D_F1a, I_F3;6). However, this means that the requirements need not (some of the sources even say should not) be too detailed (I_A;5,6,11, I_C1a;6, P_F1a; D_F1a). In case A, the development organization explicitly asked sales people for “killing arguments” only, not a detailed list of requirements (I_A;5). This, combined with the experience and understanding of the existing systems, makes a detailed list of requirements superfluous (i.e. during these early activities; later a formal requirements specification may be required). The people devising the vision of the future system (e.g. a small evaluation group)
need to study the other systems, preferably live (I Cá:6, DÉ2a, IF3:6). Case F2 involves complex scientific physics calculations, and the study of the existing systems’ documentation of the implemented models was an important activity (IF2e:6, IF2f:6). When looking at the state of the existing systems, an open mind for other solutions than the current way of doing things is essential (IF3:11).

We identified one practice associated with the long time scale involved in integration:

**Proposition III: Improve the current state.** To gain acceptance, the efforts invested in the integrated system must not only present the same features as the existing system, but also improve the current state.

The existing systems must be taken into account (see practice “Collect experience from existing systems”), but one should not be restricted by the current state (IF2f:6); in case F2, it was indeed considered a mistake to keep the old data format and adapt new development to it (IF2a:9, IF2d:7, 9, 11). The actual needs must be more important than to preserve the features of the existing systems (IF3:11). One interviewee stated that a new system would take ~10 years to implement, and a merged (and improved) system must be allowed to take some years as well (IF2f:6). In case E1, integrating several small, separate pieces as was envisioned required a more structured language (Ada), even though it would in principle be possible to reuse many existing parts as they were written in Fortran (IE1:6); the organization was interested in Ada as such, which also contributed to this choice (IE1:7).

The remaining two practices are not unique to in-house integration. In fact, they should be common sense in any decision-making in an organization. The reason to list them explicitly is that they were mentioned repeatedly in the cases, partly based on mistakes in these respects.

**Proposition IV: Timely decisions.** Decisions must be made in a timely manner (I Cá:6, I CB:6, 11).

When no decisive technical information has been found, a decision should be made anyway. In case C, the decision to discontinue one of the systems could have been made much earlier, as no new important information surfaced during the endless meetings with the small technical group (I CB:6). This means that one year of development money was wasted on parallel development, and the discontinued system has to be supported for years to come (I Cá:6, I CB:6). “It is more important with a clear decision than a ‘totally right’ decision” (I CB:11). You cannot delegate the responsibility to agree to the grassroots (I CB:6). “Higher management must provide clear information and directives… It is… unproductive to live in a long period of not knowing” (I CB:11).

**Proposition V: Sufficient analysis.** Before committing to a vision, sufficient analysis must be made – but not more.

Obvious as that may seem, the difficulty is the tradeoff between the need for understanding the existing systems well enough without spending too much time. In case F2, insufficient analysis caused large problems: what was believed to involve only minor modifications resulted in complete re-design and implementation (IF2a:9, IF2b:9, IF2c:3, IF2d:6, 11). One method of ensuring sufficient analysis could be to use the “small evaluation group” practice. Of course, pre-decision analysis somewhat contradicts the practice “timely decisions”; a stricter separation from the actual implementation process is also introduced, implying a more waterfall-like model which might not be suitable (IF1b:5, 6).
4 Architectural Compatibility

Architectural compatibility is a largely unexplored area, in spite of the fact that it is a widely recognized problem [22,23,28]. More is known of problems than of solutions. In this section, we elaborate the notion of compatibility by investigating what was actually reused in the cases. The conclusions are thus of a practical nature (what seem to make sense to reuse under what circumstances) rather than a precise definition of compatibility.

First, we present a framework for discussing reuse (sections 4.1 and 4.2), followed by a number of observations based on the cases (section 4.3).

4.1 What Software Artifacts Can Be Reused?

Although software reuse traditionally means reuse of implementations [47], the cases repeatedly indicate reuse of experience even if a new generation is implemented from scratch, i.e. without reuse of code. In order to capture this, we have chosen to enumerate four types of artifacts that can be reused: requirements, architectural solutions (structure and supporting framework mechanisms), components and source code. The first two means reuse of concepts and experiences, and the two latter reuse of implementations.

- **Reuse of Requirements.** This can be seen as the external view of the system, what the system does, including both functionality and quality attributes (performance, reliability etc.). Reusing requirements means reusing the experience of features and qualities that have been most appreciated and which need improvement compared to the current state of the existing systems. (Not discussed is the aspect of how the merge itself can result in new and changed requirements as well; the focus here is on from which existing systems requirements were reused.)

- **Reuse of Architectural Solutions.** This can be seen as the internal view of the system. Reusing solutions means reusing experience of what have worked well or less well in the existing systems. With architectural solutions, we intend two main things (for more details see 4.4):
  - **Structure** (the roles of components and relations between them), in line with the definition given e.g. by Bass et al [5]. Reusing structure would to a large part explicitly recognize architectural and design patterns and styles [1,12,27,81].
  - **Framework.** A definition suitable for our purposes is an “environment defining components”, i.e. an environment specifying certain rules concerning how components are defined and how they interact. A framework embodies these rules in the form of an implementation enforcing and supporting some important high-level decisions.

- **Reuse of Components.** Components are the individual, clearly separate parts of the system that can potentially be reused, ideally with little or no modification. We use the term “component” in a wider sense than in e.g. the field of Component-Based Software Engineering [87]; modules seem to be the appropriate unit of reuse in some systems, and hardware nodes (with embedded software) in others, etc.

- **Reuse of Source code.** Source code can be cut and pasted (and modified) given the target programming language is the same. Although it is difficult to strictly distinguish between
reusing source code and reusing and modifying components, we can note that with source code arbitrary chunks can be reused.

For each of these levels, there are associated documentation that could or would be reused as well, such as test plans if requirements are reused, and some user documentation if user interface components are reused.

For a large, complex system, the system components can be treated as sub-systems, i.e. it is possible to discuss the requirements of a component, its internal architectural solutions, and the (sub-) components it consists of, and so on (recursively). If there are similar components (components with similar purpose and functionality) in both systems, components may be decomposed and the same reasoning applied to the components. We can thus talk about a hierarchical decomposition of systems. Large systems could potentially have several hierarchical levels.

Reusing, decomposing and merging components means that the interfaces (in the broadest sense, including e.g. file formats) must match. In the context studied, where an organization has full control over all the systems, the components and interfaces may be modified or wrapped, so an exact match is not necessary (and would be highly unlikely). For example, if two systems or components write similar data to a file, differences in syntax can be overcome with reasonable effort, and the interfaces can be considered almost compatible. However, reuse of interfaces also requires semantic compatibility as well as preservation (or compatibility) of non-functional properties, which is more difficult to achieve and determine. The semantic information is in most cases less described and assumes a common understanding of the application area.

Although reuse of all artifacts is discussed, the focus is on reuse of architectural solutions and components, and on the recursive (hierarchical) decomposition process.

### 4.2 Possible Basic Types of Reuse in Software Merge

For each artifact enumerated, it is possible to apply the high-level strategies presented earlier: *Merge, Choose One, and Start from Scratch*; for the purpose of this chapter we would like to rephrase them as a) reuse from both existing systems, b) reuse from only one of the existing systems, and c) reuse nothing. (The *No Integration* strategy is always applied to the whole system, so discussing it per artifact would not contribute to the discussion.) As for the high-level strategies, we discuss the situation of more than two systems only in connection to the cases. A matter of interpretation is where to draw the border between type a, “reuse from both”, and b, “reuse from one”, in the situation when only very little is reused from one of the systems; this is discussed in the next section for some cases.

Different types of reuse can be applied at each of the above mentioned/enumerated artifacts. For example, requirements might be reused from all systems (type a), but only the architecture and components of one is evolved (type b). This makes it possible to search for certain patterns in the cases revealing how different types of reuse for different artifacts are related. For example, is it possible to reuse architectural solutions from only one of the existing systems but reuse components from both? If so, under what circumstances?

### 4.3 Observations Concerning Reuse in the Cases

The cases are summarized in Table 3. For cases A, B, C, E1 and F3 we can show the single system that was the outcome, so for the system represented by these columns, we present the
type of reuse from the earlier system. For the other cases, the original systems are still evolved and deployed separately, and we report the reuse expected in the envisioned future system as well as the current cross-reuse from the other system(s). In addition, there is a possibility to recursively look into components and consider the requirements, architectural solutions, and (sub-) components of the components, etc. This is done for cases D and F2 where we have enough material to do so (for case F2 in two levels); for most of the others, this did not make sense when there was no reuse from more than one system.

In the table, for each system we have listed the four artifacts considered (requirements, architectural solutions, components, and source code) and visualized the type of reuse with black for reuse of type a “reuse from all”, dark grey for reuse of type b “reuse from one”, and light grey for reuse of type c “no reuse”. Fields that have not been possible to classify unambiguously are divided diagonally to show the two possible alternative classifications; these fields have been marked with a number indicating a text comment (to be found below). (The classification has been made by the researchers jointly, without disagreement.)

Based on Table 3, we can make a number of observations:

**Observation 1.** A striking pattern in the table is the transition when following a column downwards from black to dark grey to light grey, but not the other way around (not considering transitions between components and source code). This means that:

*If it is not possible to reuse requirements from several of the existing systems, then it is difficult, if not impossible, or makes little sense to reuse architectural solutions and components from several systems.*

and:

*If it is not possible to reuse architectural solutions from several of the existing systems, then it is difficult, or makes little sense to reuse components from several systems.*

There is only one possible exception from these general observations (system F2:2D, see comment 5). We can also note that the type of reuse of architectural solutions very often comes together with the same type of reuse for components. This means that if architectural solutions are reused, components are often reused. This makes sense, as the prerequisites for reusing the components are then met, and it would often be a waste not to reuse existing implementations.

**Observation 2.** In the cases where “reuse from all” occurred at the architectural solutions level, this did not mean merging two different architectures, but rather that the existing architectures were already similar. In the only possible counter-case (case A), the development team built mainly on their existing knowledge of their own system, adapted new ideas, and reused the concept of configurability from one other existing system. This is a strong indication of the difficulty of merging architectures; merging two “philosophies” (I_E:1), two sets of fundamental concepts and assumptions seems a futile task [28]. This means that:

*For architectural solutions to be reused from several systems, there must either be a certain amount of similarity, or at least some architectural solutions can be reused and incorporated into the other (as opposed to being merged).*
That is, the fundamental structures and framework of one system should be chosen, and solutions from the others be incorporated where feasible.

**Observation 3.** In case D and F2 where the overall architectures structure were very similar (client-server and batch sequence respectively), the decomposed components follow observations 1 and 2. This means that:

*Starting from system level, if the architectural structures of the existing systems are similar and there are components with similar roles, then it is possible to hierarchically decompose these components and recursively consider observations 1 and 2. If, on the other hand, the structures are not similar and there are no components with similar purpose and functionality, it does not make sense to consider further architectural reuse (but source code reuse is still possible).*

In the other case with similar system structures (case C) the approach was to discontinue one and keep the other, in spite of the similar structures. The reasons were: differences in the framework, the equal quality and functionality of both systems, combined with the large size of the systems. This shows that architectural structure is not enough for decomposition and reuse to make sense in practice. Nevertheless, in case C it was possible to reuse some relatively small parts from the other system (with some modification).

**Observation 4.** In cases C, D, F2 (and possibly E2) the architectures were similar. One reason for this was that in cases D and F2 the systems had a common ancestry since previous collaborations as far back as twenty years or more (I_D2a:1, I_F2a:1). There also seems to be common solutions among systems within the same domain, at least at a high level (e.g. hardware architecture); there may also be domain standards that apply (I_C6:1,7, I_F2a:1).

Although not directly based on the table, we would like to make an additional remark. When it is not possible to reuse architectural solutions or components it might still be possible to reuse and modify arbitrary snippets of source code. The benefit of this type of reuse is the arbitrary granularity that can be reused (e.g. an algorithm or a few methods of a class) combined with the possibility to modify any single line or character of the code (e.g. exchanging all I/O calls or error handling to whatever is mandated in the new framework). There seems to be a simple condition for reusing source code in this way, namely that the programming language stays the same (or maybe “are similar enough” is a sufficient condition), which should not be unlikely for similar systems in the same domain. Source code thus requires a much smaller set of assumptions to hold true compared to combining components, which require the architectural solutions to be “similar enough” (involving both structure and framework).

### 4.4 Architectural Compatibility as Part of the Vision Process

Let us now return to the vision process, and ask the question: based on the observations from the cases, what needs to be analyzed early in order to make a proper strategy selection? First, we can note that architectural compatibility is a static property of a collection of systems and cannot be negotiated if it gives an unsatisfactory answer. It is therefore essential to evaluate it properly during the vision process.

However, if a subset of the candidate systems (or some subsystems) is considered compatible it may be possible to change the scope of the integration project to include only these subsystems, thus enabling the possibility of a *Merge*. Case F1 exemplifies a change in scope, but unfortunately no suitable set of systems to merge have been found (I_F1a:1, I_F1b:1,6, I_F1c:1, I_F1d:1, I_F1e:1).
\(P_{F1a}, P_{F1b}\)). It may also be possible to evolve one or all systems towards a state in which they are compatible, i.e. performing an *Evolutionary Merge* – although, given the time required, some other strategy may be considered preferable, as shown by case C (ICa:6, ICb:6). These two approaches, change scope or synchronize the evolution of the systems, can thus be considered as means to somewhat improve the compatibility in order to make a *Merge* possible.

A definition of architectural (in-)compatibility would be subject to the same semi-philosophical arguments as definitions of architecture, and we will not attempt to provide one. Exactly what aspects of compatibility are the most important to evaluate will arguably differ for each new case. Some elements found in the cases, that we thus suggest to be analyzed during the vision process, are provided in the following. These elements can be a complement to other reports of architectural incompatibility [22,28]:

- **Structure.** Similar high-level structures seem to be required for hierarchical decomposition and component-wise comparison, a pre-requisite for a *Merge*. In case D, both systems consisted of an HMI and server, which made it possible to reuse the HMI from one system into the other. In case E2 two of the existing systems consisted of a graphical user interface (GUI) and a simulation engine, loosely coupled, which made reuse of the GUI possible. In case F2, the two existing pipe-and-filter structures were strikingly similar.

- **Frameworks.** Similarity of frameworks in the sense “environment defining components” is also one indicator of compatibility. In case F2, the framework can be said to describe separate programs communicating via input and output files. Two of the existing systems in case F3 were developed in Lotus Notes, and they were, with the words of the interviewee, “surprisingly similar” (IF3:1). In case C, the hardware topology and communication standards define one kind of framework. In case F2, the framework can be said to describe separate programs communicating via input and output files.

- **Data model.** One common source of incompatibility in systems is differences in the data model. Both syntactical and semantical differences can require vast changes in order to make system compatible. This has been a recurring problem in case F1 (IF1a:6; D_{F1a}, P_{F1a}, P_{F1b}). In case F2, a new data model was defined and the existing systems adapted (IF2e:7, IF2f:6). In case F3, the three existing systems all implemented similar workflows, however the phases were different (IF3:3).

Some systems in the cases shared a common ancestry (cases D and F2) and/or were based on common standards (C and D), but in no case were the systems compatible enough to allow for an *Rapid Merge*. This indicates that these factors in themselves do not guarantee total compatibility.

Two of the cases may serve as examples on what may happen if a *Merge* decision is made based on insufficient knowledge about the compatibility of the systems. In cases C and F1, senior management gave directions how a system merge should be achieved: “try to agree and reuse as much as possible” (ICc:6, also P_{F1a}). In case C, this caused an expensive delay as well as other problems (ICc:6,7), and in case F1 the architecture and outlined integration plan felt watered-down (D_{F1a}, IF1c:6), and nothing happened to realize it (P_{F1a}, P_{F1b}).
Retireability
Retiring a system is a difficult step, but may be necessary. As we have argued, if retiring some or all of the existing systems is considered possible, this excludes some of the high-level strategies. Trivial as this observation may seem, or even oversimplifying, this suggests that the vision process should include an analysis of retireability, meaning that various stakeholders should be asked to explicitly describe their opinion on the impact of retiring each of the existing systems. Although retireability is not a definite yes/no question, discussing it explicitly is one way of breaking down the overall task of selecting a strategy to more manageable pieces.

We only present our observations from the cases more briefly than for architectural compatibility, not because this is less important, but because the factors influencing the possibilities of retiring systems is not foremost a technical issue. Section 5.1 describes the influences found in the cases, and 5.2 provide some suggestions on how retireability should be evaluated as part of the vision process.

5.1 Observations from the Cases
Retireability, unlike architectural compatibility, can be reevaluated or renegotiated. While all cases considered the retireability of their existing systems it appears as this was often not done explicitly to the same extent as the evaluation of compatibility.

The life cycle phase of the existing systems should be considered (ICb:1,7, ICb:6, IE1:4, IF2e:6, IF2a:3). Case C may serve as a negative example of this, where a new generation of both existing systems was being developed, but not yet released, and the obvious choice would seem to be to discard either of them before release (ICb:7, ICb:6); however development did continue until both systems were released, which led to lots of extra costs and problems (ICb:6).

Another implication of the life cycle phase of the existing systems is what potential they have to be further evolved in the future. In case D, one of the existing HMIs had been restructured a few years before the company merger, from a monolith which was very difficult to maintain without understanding the complete system, into a component-based system, which has made it “straightforward to implement new functionality” (IDb:7). This was influential when deciding to keep this HMI (IDb:7,8). In case B, the loose integration of one of the systems which would ensure long-term evolvability, was influential when choosing system (IB:7,9,10,13).

Another influence is satisfaction with existing systems. This involves many stakeholders (architects, users, management, etc.) and many aspects (functionality, quality, architecture, performance, modifiability/evolvability, etc.). When one or more of the existing systems are considered unsatisfactory there is a tendency to favor replacing the unsatisfactory system(s). If some of the existing systems are considered aged, they are candidates for retirement, as in case D where one of the existing HMIs was considered aged and was replaced by another (IDb:3). In case F2, one of the sites was about to develop something new, while the other had realized some fundamental problems with the physical models their software embedded, which led to a successful common development project (IF2e:6, IF2a:3).

We may note the difference between retiring implementations (which this paper discusses) and the message communicated externally to customers and users. In case C, there was a big
difference in the message to the market (that the products would be merged into one product family) and internally (essentially Choose One system and retire the other) (ICa:6).

5.2 Retireability as Part of the Vision Process

Changes are usually met with some reluctance, and it is not easy to start discussing retiring a system that is in use. Moreover, retireability is not easily stated as a yes or no question. The effects of retirement (followed by replacement) should be investigated from various perspectives (of e.g. users, customers, marketing department). From the users’ point of view, proven high-quality systems are not easily discarded, while systems considered aged are candidates for retirements. Retiring a system might also effectively mean that staff on one site will loose their jobs, which raises an additional ethical and economical dilemma.

Some of the negative effects of retiring might be possible to handle, such as providing seamless migration solutions for customers, and marketing the replacing system as a natural successor (ICa:6). The final decision will involve weighing the negative consequences of retiring against the positive consequences of being able to Choose One or Start from Scratch (which are only possible if some systems are planned for retirement).

Retireability should be considered from many different stakeholders’ point of view, as during any requirements engineering activity in any context [45,78]. However, it is likely that different people will have different opinions, and there must be mechanisms and roles that ensure that a timely decision is made, weighing all the different benefits and drawbacks of retiring the particular systems. Our suggested practice “Small Evaluation Group” could be organized so that different stakeholders are involved at different stages. They would evaluate and compare the existing systems from different point of view, formulating high-level requirements on the future system in terms of the functionality and quality of the existing systems. In case F1, users, architects/developers and managers evaluated the systems in a defined process with three phases (DF1, [55]).

6 Implementation Process

When selecting a strategy, one must understand the consequences on the time, cost and risk of implementing it. It should be expected that the implementation process will be different depending on the strategy chosen.

We first present some recurring patterns among the cases, divided into risk mitigation tactics (section 6.1) and process practices (section 6.2). In section 6.3 we use these observations to suggest how the implications on the implementation process should be used when selecting a strategy during the vision process.

6.1 Suggested Risk Mitigation Tactics

We found five recurring risk mitigation tactics. These seem to come mainly from the fact that there are two (distributed) groups involved; these practices are thus recommendable to every distributed software development effort. They are very much in line with recommendations from other research in the field of distributed software development [13,14,41]. Although they are not unique to in-house integration, we present them here because they are directly based on the experiences from the cases. This might suggest that they are especially important in the in-house context; one explanation is that when some systems or system parts are even remotely considered for retirement, the people and organization behind those systems will react against it.
Proposition I: Strong project management. To run integration efforts in parallel with other development efforts, a strong project management is needed (e.g. $I_{F1c}:9,11$, $I_{F2b}:5,11$, $I_{F2e}:9,11$).

To be able to control development, senior management and project management must have, and use, economical means of control. Funding must be assigned to projects, and cut from others, in a way that is consistent with the long term goals; otherwise the existing systems will continue to be developed in parallel with some sub-optimal goal ($I_{C6}:11$, $I_{D6}:3$, $I_{D6}:6$, $I_{F1b}:11$). In case C, not until economical means of control were put into place did development of the system-to-be-discontinued stop ($I_{C6}:6$), and in case D assignment of development money to delivery projects is apparently not in line with the long-term integration goals ($I_{D6}:3$, $I_{D6}:6$).

“Projects including costly project specific development are not punished. The laws of market economy are disabled.” ($I_{D6}:3$) Case E1, a cooperation led by a research institute, can serve as a counter-example. Here, enthusiasm apparently was the driving force, and the lack of strict management was even pointed out as contributing to success ($I_{E1}:9,11$). Although we agree it is essential to create a good and creative team spirit [24], we believe it would be bad advice to recommend weak or informal project management, at least for larger projects.

Proposition II: Commitment. To succeed, all stakeholders must be committed to the integration, and management needs to show its commitment by allocating enough resources (e.g. $I_{F1b}:11$, $I_{F1c}:11$).

In case F2 it was pointed out (based on negative experience) that for strategic work as integration is, one cannot assign just anyone with some of the required skills; the right (i.e. the best) people must be assigned, which is a task for project management ($I_{A}:11$, $I_{F2b}:11$, $I_{F2a}:9,11$, $I_{F2e}:9,11$). Realistic plans must be prepared, and resources assigned in line with those plans ($I_{F1c}:11$). When directives and visions are not accompanied with resources, integration will be fundamentally questioned ($I_{F1b}:3$, $I_{F1c}:6,9$). When there is a lack of resources, short-term goals tend to occupy the mind of the people involved. The motivation for integrating is to reduce some problems in the longer term rather than producing some customer value, which means that (lacking external pressure) the internal commitment becomes more important than otherwise.

Proposition III: Cooperative grassroots. In order to succeed, the “grassroots” (i.e. the people who will actually do the hard work) must be cooperative, both with management and each other.

The people who will perform all the tasks required to implement the integration plan are divided into two (or more) physically separated groups, who may see each other as a threat. It may be difficult to motivate people for tasks that they feel are stealing time from their ordinary work, and that even might undermine their employment. In case D, the grassroots considered explicitly whether cooperation was of benefit to themselves ($I_{D6}:6$); they decided that for cooperation to succeed they needed to show they were willing to build trust, that they had no hidden agenda ($I_{D6}:6,11$). The overall goals must be made clear to the grassroots to gain the necessary commitment and “buy-in” ($I_{C8}:11$, $I_{F1b}:11$). The “not invented here syndrome” is dangerous for cooperation ($I_{D6}:6$, $I_{F1c}:11$). Case E1 illustrates that a project with “enthusiasm, lively discussions and fun people” may drive the integration so that the need for strict management project schedules is reduced ($I_{E1}:9$); what contributed most to success were the fun people and the lack of strict management ($I_{E1}:11$).
Proposition IV: Make agreements and keep them. To be able to manage and control a distributed organization formal agreements must be made and honored.

In case F2, it was pointed out as a big problem that requirements and design evolved driven by implementation (I_{F2b}:6, I_{F2c}:9, I_{F2d}:6, 11). Even in the informally managed case E1, the importance of agreeing on interface specifications and keeping them stable was emphasized (I_{E1}:7,9). When you do not meet the people you work with in person very often, and there are local tasks on both sides that easily gets prioritized, more formalism than usual is required. You must have agreements written down and then stick to them (I_{F1c}:9,11).

Proposition V: Common development environment. To be able to cooperate efficiently, a common development environment is needed (P_{F1b}, I_{F2b}:6,11, I_{F2e}:11,12, I_{F2f}:12). With “development environment” we include e.g. development tools, platforms and version control systems. In case F2, it was difficult to synchronize the efforts (I_{F2e}:11); e.g. source code was sent via email and merged manually (I_{F2b}:6). In case F1, the difficulties of accessing the other site’s repository caused an unnecessarily long period of (unknowing) parallel development (P_{F2b}).

6.2 Suggested Implementation Process Practices

The long time scale typically implied by integration gives rise to the problem of keeping people motivated and management committed. There is a constant tension between the local priorities of the existing systems and the long-term goal of an integrated system. Without a minimum effort in integration, the environment and the vision will change more rapidly than the integration makes progress, which means only a waste of resources. To address this, we observed a number of practices:

Proposition I: Achieving momentum. Integration cannot be sustained by external forces indefinitely, but mechanisms must be put into place that provides internal converging forces (e.g. I_{F2e}:9).

If the Evolutionary Merge strategy is chosen, it must be ensured that changes made to the systems are done in line with the integration goal. These changes often compromise the individual systems’ conceptual integrity, and moreover often require more effort. The challenge is to identify and implement internal converging forces, so that as changes are made, it becomes more and more efficient to do other changes that also contribute to the long-term Evolutionary Merge; in this manner the integration will gain a certain momentum [24] and partly drive itself. Such an internal converging force could be the development and use of common libraries that are superior in some way than the existing ones (I_{F2e}:7,12). The opposite would be when external forces are constantly needed to achieve convergence, such as heavy-weight procedures which’ purpose is not understood by the developers. This will take a lot of energy from the staff and the organization, will create stress and tension, and may also lead to a recurring questioning about the purpose of integration (I_{F1b}:3,11, I_{F1c}:6,9). One of the interviewees in case F1 (which has not made significant measurable progress during the 4 years that have passed since the company merger) asked “from where comes the driving force?” (I_{F1c}:9), pointing at the fact that integration is not a goal in itself. A balance between convergence and divergence must be found; divergence could be allowed in order to develop customer-specific systems in parallel, if there are mechanisms that will enforce standardization and convergence from time to time (I_{B}:7,11,13). (These terms: converge, diverge, driving force, momentum, were terms used by many of the interviewees themselves).
Proposition II: Stepwise delivery. Ensure early use and benefit of the value added in the transition process.

Typically, the vision lies far into the future, and integration processes are less predictable than other development projects (I_F2c:10,12). Maintaining the long-term focus without monitoring and measuring progress is impossible (I_A:6,9, I_B:1, I_D8:12, I_D9:6, I_F1b:6, I_F2c:6,11, I_F2f:6). A waterfall model might be inappropriate, as the system runs the risk of not being feasible at time of delivery (I_F1b:5,6); there is a too long time to return of investment (I_B:1). Closely associated is the approach of a loosely integrated system: an integration point should be found and all subsequent activities, although run as separate delivery projects, will little by little make integration happen (I_B:6,7, I_F1b:6,7,8,11; the proposed integration point in case F1 was a data storage format). There is however a tradeoff to be made, as there are typically some common fundamental parts (such as infrastructure) that need to be built first (P_F1a, D_F1a, I_F2c:7). In contrast to development of new products, or new product versions, these activities are performed in parallel and often not considered the most important. For these reasons the decisions regarding the implementation process do not only depend on the process itself, but also on many unrelated and unpredictable reasons. Stepwise deliveries and prototyping have been used for new development to increase process flexibility, could be one way of achieving the desirable momentum. This was also a recurring opinion among the interviewees, with some variations:

- **User value.** Some of the interviewees maintained that there must be a focus on deliveries that gives user value, and a clearly identified customer (I_B:1,7,11,13, I_F1b:6,11). If it is possible to utilize a customer delivery to perform some of the integration activities, this will be the spark needed to raise the priority, mobilize resources, gaining commitment etc. (I_F2c:6,11). However, it should also be noted that customer delivery projects typically have higher priority than long-term goals such as integration, and may subtract resources and commitment from the implementation process. The extreme would be to focus only on immediate needs, questioning the need of integration at all (I_F1b:3,11, I_F1c:6,9).

- **Prototyping.** The *Start from Scratch* strategy is essentially new development, which is the typical situation where prototyping would be a way of reducing risk; in Case A, a prototype was developed as a way to show an early proof of concept (I_A:1,6,9,11). In the case of *Choose One*, prototyping could mean making a rapid throw-away extension of the selected system to test how well it would replace the other(s). For a *Merge*, prototyping would not mean so much demonstrating functionality as investigating the technical incompatibilities and some quality aspects of the merged system. When merging the 3D simulators of case F2, one internal release was planned where robustness and performance was prioritized away, in order to more rapidly understand how well the different modules would fit together.

- **Do something concrete.** In some cases where it has been difficult to formulate, or agree on, or commit to a vision, the opinion has been raised that it is better to move on and do something that is useful in the shorter term, e.g. implement some specific functionality that is useful for both systems. This is then be used as a learning experience (I_F2c:11, I_F2c:6). However, there is also a potential danger that implementation will drive requirements. This happened in case F2 where requirements and design evolved uncontrolled as implementation continued (I_F2c:6, I_F2c:9, I_F2c:6,11); it would have been better to either freeze the requirements or to use a development model that is better suited to allow for constant changes to requirements and design.
The practice of stepwise delivery implies that there will be iterations between the vision process and the implementation process.

### 6.3 Considering Implementation Process in Vision Process

The characteristics of the implementation phase affects the overall success of integration, and should be carefully evaluated during the vision process. Although this must be evaluated individually for any specific case, in general Merge seems to be more complex than the Choose One and Start from Scratch strategies. This is partly because these latter strategies can be expressed in terms of retiring a system, maintaining or evolving a single system, and developing and deploying a new system, while a Merge is less familiar to software organizations. Also, it seems that the risk mitigation tactics and process practices found become more important in case of a Merge. This is partly because the organization commits to a long-term distributed development activity, while Choose One or Start from Scratch do not inherently demand distributed development. In practice they will however involve a certain amount of distributed collaboration, e.g. to transfer knowledge and possibly staff.

With a Merge, it is also arguably more difficult to “achieve momentum” due to an ever-present tension between focusing on long-term goals and short-term goals, between global goals for the organization and local for the previous self-standing departments in charge of their system (this depends on how drastic the organizational changes after a company merger have been). Although this tension seems to always be present in any software development or maintenance activities, including the Choose One and Start from Scratch strategies, Merge is more complex as it involves two systems instead of one, two systems that need to be evolved simultaneously and prioritized in the same manner. This increases the importance of making formal agreements (risk mitigation tactic “make agreements and keep them”) dramatically.

### 7 Resources, Synchronization, Backward Compatibility

This section provides some observations on additional issues that need to be considered during in-house integration, and explicitly analyzed during the vision process. As for the section on retireability, the short descriptions do not mean we consider these issues of little importance, only that they have not been in focus of our research.

Availability of resources, such as time, money, people, and skills, has a big influence on the choice of strategy. Fundamentally, the architect and the organization must ask whether a certain strategy can be afforded. Even if the expected outcome would be a common, high-quality system, the costs could simply be prohibitive. In case E2, resource constraints resulted in some integration of two existing systems, and the retirement of the third system without replacement (I_E2:13, D_E2a).

The relation to other development activities must also be considered. As integration has to be done in parallel with the ordinary work within the organization, this often leads in another direction (I_F1a:9). There is a need to synchronize all parallel development efforts within the company, otherwise projects run too freely and “sub-optimal solutions” are created (I_F1c:6).

Another issue that needs to be considered is the need for some type of backward compatibility with the existing systems, for example by supporting existing data formats or providing data migration mechanisms tools (I_C3:6, I_F2a:5, P_F1). Not only existing data but also existing user processes must be considered – in order to achieve the rationalization goals of integration, it may be necessary to require (some of) the users to change their way of working (I_F1b:3, I_F2:6). Case F3 may serve as a positive example where users understood and accepted that they had
to change their processes somewhat, and were willing to do this as they understood the overall benefits of having a common system – although they wanted to have the future system processes as similar to their existing ones as possible (I_F3:3).

8 Analysis

This section starts by summarizing the cases, arguing that the concepts and terms introduced are useful to explain the events in the cases (section 8.1). We then suggest a procedure for exploring and evaluating different strategies, synthesizing the observations of the different selection criteria discussed separately in previous sections (section 8.2).

8.1 Strategy Exclusion and Selection in the Cases

Table 4 summarizes which strategies are excluded according to our reasoning in this paper, based on table 2 and our interpretation of the cases. Exclusion is marked with black, with a question mark where the classification is open for discussion; we have chosen to show the interpretation that could falsify our proposed scheme, i.e. excluding the most strategies. In case C retireability was clearly renegotiated, and in case C and F1 the decision changed, illustrated with multiple entries for these cases showing these iterations. There are also entries for the constituent components of cases D and F2, where the Merge strategy was chosen and currently implemented. A check mark indicates which strategy was finally selected and implemented (cases A, B, C (final), D_HMI, E1, E2, F2_Pre, F2_Post, and F3), or desired (for the cases not yet finished, indicated with an asterisk, and for cases C and F1 where the decision was later changed).

As the tables visualize, Choose One or Start from Scratch was chosen in favor of Evolutionary Merge whenever possible in the cases; the only case where Evolutionary Merge was chosen was when there was no other option (case F2_3D). This supports our reasoning that Merge is perceived as the most difficult strategy to implement. Out of the six rows where both Choose One and Start from Scratch remained, Start from Scratch was chosen in five, which might indicate a common wish to take the opportunity in this situation for a complete remake and invest in a new generation of the system(s).

As compatibility is not re-negotiable, and has such profound impact on the possible integration strategies, it must be carefully evaluated and communicated prior to a decision. Obvious as this may sound, the cases illustrate that this is not always the case. In case C, management insisted on a Rapid Merge, although considered impossible by the architects (I_Ca:6, I_Cb:6) resulting in several hundred person-years being lost. In case F1 an Evolutionary Merge was decided upon because the systems could not be retired, even though the systems were incompatible (I_F1c:6, D_F1a, P_F1a), resulting in no progress after 4 years of work. The decisions were, when considered in isolation, perfectly understandable: it is easier to not bother about the complexities associated with retiring systems, and it is easier to assume that technicians can merge the systems. This is a typical trade-off situation with no simple solution.

It is of course equally important to evaluate the possibilities of retiring the existing systems, but it is very difficult for us as outsiders to evaluate whether the decisions in the cases were good or bad, and we will avoid doing that. We can nevertheless point at the fundamental problem encountered when the existing systems are considered impossible to retire, but are at the same time totally incompatible – there is simply no integration solution, as illustrated by case F1. Case C shows the same difficulty: although the systems were somewhat compatible their sheer size seemed to for all practical reasons exclude Evolutionary Merge.
8.2 Suggested Analysis Procedure

All these observations taken together allow us to suggest a checklist-based procedure for refining and evaluating a number of alternatives. We are not proposing any particular order in which the strategies should be considered, neither of the activities suggested for each. The activities could very well be carried out in any order, in parallel, or iteratively, continuously putting more effort into refining the most crucial analyses in order to make as well-founded decision as possible.

Starting at the highest level with existing systems $X$, $Y$, $Z$, etc., the pros and cons of each strategy should be considered and documented.

- **Start from Scratch**
  - Consider the impact of retiring all the existing systems. (See section 5.)
  - Outline an implementation plan and consider the associated cost and risk. This plan must include development and deployment of the new system as well as the parallel maintenance and eventual retirement of the existing systems. (See section 6.)

- **Choose One:** Assuming that system $X$ would be chosen, do the following (then do the same assuming that the other systems $Y$, $Z$ etc. would be chosen):
  - Consider the impact of retiring the other systems. (See section 5.)
  - Estimate how well system $X$ would replace the other systems.
  - Outline an implementation plan and consider the associated cost and risk. This plan should include evolution and deployment of system $X$ as well as the parallel maintenance and eventual retirement of the other systems. (See section 6.)

- **Merge:** Identify incompatibilities, and if possible decompose hierarchically:
  - Compare the systems regarding at least 1) the high-level structures and component roles, 2) the frameworks, and 3) their data models. If they are similar, decompose the system(s) into components, and repeat the procedure for each pair of components $\alpha_X$ from system $X$, component $\alpha_Y$ from system $Y$, etc. Otherwise, *Merge* is very likely the least cost-efficient integration strategy. (See section 4.)
  - Outline an implementation plan and consider the associated cost and risk. This plan should include stepwise deliveries of the existing systems, and take into account the parallel maintenance and evolution of the existing systems. (See section 6.) (We can note that the activities suggested here are identical for both Evolutionary and Rapid Merge; the only difference would be how much time is estimated in the plans.)

For all of the strategies, also consider other things that may influence the selection: resources, synchronization, and backward compatibility (section 7).

The result of this procedure will be a set of alternatives of what the integrated system could look like, with associated benefits and drawbacks along many dimensions (the features of the actual system, implementation time, cost and risk, negative effects of retiring systems, etc.). When following this procedure, some alternatives will likely be immediately discarded (and it
makes no sense to elaborate those alternatives exhaustively in the first place). A trade-off decision will be required to finally select the overall optimal alternative, where expected pros and cons are weighed against each other.

(It should be noted that we have by purpose avoided over-specifying this procedure, as we believe there are many different practices in different organizations, which might all be applicable. We therefore do not want to mandate a certain sequence of activities, neither do we want to formalize how to assign weights or how to document the outcome.)

9 Related Work
The insight that software evolves, and has to evolve, is not new [11,62,72,73]. Software has to be extended in different ways to keep up with evolving needs and expectations, and interoperability and integration is one type of extension. However, the topic of in-house integration has not been previously researched. In our previous literature survey [51], we found two classes of research on the topic of “software integration”:

1) Basic research describing integration rather fundamentally in terms of a) interfaces [36,92,93], b) architecture [5,29,33], architectural mismatch [28], and architectural patterns [12,27,81], and c) information/taxonomies/data models [30,30,71,85]. These foundations are directly applicable to the context of in-house integration.

2) There are three major fields of application: a) Component-Based Software Engineering [20,68,87,91], including component technologies, b) standard interfaces and open systems [68,69], and c) Enterprise Application Integration (EAI) [21,79]. These existing fields address somewhat different problems than in-house integration:
   i) Integration in these fields means that components or systems complement each other and are assembled into a larger system, while we consider systems that overlap functionally. The problem for us is therefore not to assemble components into one whole, but to take two (or more) whole systems and reduce the overlap to create one single whole, containing the best of the previous systems.
   ii) These fields typically assume that components (or systems) are acquired from external suppliers controlling their development, meaning that modifying them is not an option. We also consider systems completely controlled in-house, and this constraint consequently does not apply.
   iii) The goals of integration in these fields are to reduce development costs and time, while not sacrificing quality. In our context the goals are to reduce maintenance costs (still not sacrificing quality).

There are also methods for merging source code [7,66], and even architectural descriptions [89]. The focus is on merging development branches saved in version management system. However, when integrating large systems with complex requirements, functionality, quality, and stakeholder interests, the abstraction level must be higher. These approaches could possibly be useful when discussing the **Merge** strategy, if the existing systems have very similar structures.

As system evolves and ages, a common observation is that they deteriorate, degrade, or erode [72,90]. Refactoring relates to the systematic reorganization of the software in order to improve the structure [26]. It is likely that the existing systems in an in-house integration situation have degraded somewhat, and that refactoring may be an additional activity for the **Choose One** strategy, or prior to a **Merge**.
Software architecture is defined in academia in terms of “components” (or “entities”) and “connectors” [5,29,74], which is possible to formalize [1,3] and have resulted in catalogues of generally useful structural patterns [12,27,81]. We have adopted this structural perspective in the present work, but also believe there is more to software architecture than structure: we saw frameworks (in the sense “environment defining components”) and data models as sources of (high-level) incompatibilities. A software architect is typically concerned with much more than structural diagrams and formalisms, and is often considered being the person who understands the language and concerns of other stakeholders [83,94], and/or the person who monitors and decides about all changes being made to the system to ensure conceptual integrity and avoid deterioration [72,90]. We believe many of tasks outlined in the present paper matches this job description well. The field of Component-Based Software Engineering is closely related, with focus on how to build systems from pre-existing components [20,87,91].

There are several proposed methods for architectural analysis, such as the Architecture Trade-Off Analysis Method (ATAM) and the Cost-Benefit Analysis Method (CBAM) [15]. Although primarily designed to be used during new development, they have been used during system evolution [15,42,44], and could very well be used to evaluate alternatives of a future integrated system.

Closely related to our description of architectural compatibility is the seminal “architectural mismatch” paper, which points out issues to be assessed as part of the architectural compatibility [28]. The “composability” of components views a similar problem from the view of components, which need to be interoperable and complementary to be composable [67]. Also related to assessing architectural compatibility are architectural documentation good practices [16,33,37].

For new development, there are a number of established software development models: the traditional sequential waterfall model with different variants [65], iterative, incremental, and evolutionary models [8,65], the commercially marketed Rational Unified Process (RUP) [46] and recently agile methodologies [6,84]. There is a body of research specifically covering the context of distributed software teams [13,14,32,43], although not concerning the specifics of in-house integration. There are also literature covering good practices, some of which overlap with our practices extracted from the cases (e.g. on commitment [2]). The most well-known compilation of so-called best practices for software development in general is arguably the Capability Maturity Model Integrated (CMMI) [17]. However, the main focus of the available knowledge is new development, and to some extent other activities such as evolution, maintenance [4], deployment, and product integration [61]. There is some research also in non-classical new development models related to in-house integration, such as component-based development processes [19,38,91], and concerning reuse [40,76]. Although there is certainly some overlap, the existing literature cannot be directly applied to in-house integration, where we have seen that Merge in particular is difficult to formulate in terms of existing processes, and that the vision process itself contains some elements unique to in-house integration.

Many issues are not purely technical but require insight into business, and many decisions require awareness of the organization’s overall strategies. Strategic planning (and strategic management) is known from business management as a tool for this kind of reasoning, that is to systematically formulate the goals of the organization and compare with the current and forecasted environment, and take appropriate measures to be able to adapt (and possibly control) the environmental changes [18,88]. In our case, investigating retireability clearly fits
within the framework of strategic planning, by explicitly considering the money already
invested, existing (dis)satisfaction, risk of future dissatisfaction, estimated available resources,
and weigh this based on the perceived possible futures. In fact, the whole process we have
described, and perhaps much of an architect’s activities should be cast in terms of strategic
planning such as the PESTEL framework or the Porter Five Forces framework [75]. (It should
perhaps be noted that our term “integration strategy” is a plan, which is not synonymous to a
company strategy in the sense of strategic planning.)

10 Summary

In-house integration is a complex and difficult undertaking, which might nonetheless be
absolutely necessary for an organization as its software systems are evolved and grow, or after
company mergers and acquisitions. This topic has not yet been addressed in research, so the
proper starting point is a qualitative study. This paper presents a multiple case study
consisting of nine cases in six organizations, where the data sources include interviews (in all
cases), documentation (in four cases), and participation (in one case). The organizations and
systems are of different types and sizes, ranging from a maintenance and development staff of
a few people to several hundred people, and all have a significant history of development and
maintenance. The domains of the systems included safety-critical systems (two cases),
physics simulations (three cases) and different types of data management (four cases). We
consciously avoided pure enterprise information systems as there are existing
interconnectivity solutions for this domain (although we believe our findings are applicable
also to that domain). Based on the cases we suggest that the integration activities should be
considered as two processes: a vision process and an implementation process.

Certain recurring practices for the vision process were identified in the cases, needed because
of some of the characteristics of in-house integration: typically no single person in the
organization knows all the existing systems well. Since it is a big and long-term commitment
risk must be reduced by involving the right people at the right point in time, in order to have a
sufficient basis for making a good decision, while making the evaluation as rapidly as
possible. The practices found have been labeled Small evaluation group, Collect experience
from existing systems, Improve the current state, Timely decisions, and Sufficient analysis.

The goal of the vision process is to select a high-level strategy for the integration. We have
named the extreme alternatives Start from Scratch, Choose One, and Merge. To evaluate the
possibilities of a tight Merge, the architectural compatibility of the systems must be
evaluated. The findings in the cases strongly suggest that the high-level structures of the
existing systems must be similar for a Merge to be possible. Fortunately, we have found that
within a domain it is not unreasonable to expect the systems to have similar structures due to
standards (formal or de facto standards). If the structures at a high level are similar (e.g.
client-server), it becomes possible to hierarchically decompose the system by looking into
each pair of components (i.e. the clients and servers separately) and considering whether their
internal structures are similar enough to enable picking components. The framework in which
the system is implemented, in the sense “environment defining components” also must be
similar, which again is not uncommon among systems built in the same era in the same
domain. The other major source of incompatibilities in the cases was differences in the data
model. The Merge strategy can be subdivided into two types, Rapid and Evolutionary,
distinguished by the time required to merge. Based on our cases, although Rapid Merge is
sometimes demanded by management, it is typically considered unrealistic by the technicians
who understand the architectural incompatibilities better. This is a lesson to consider in future
projects, the danger of management underestimating the technical difficulties.
There are clearly many other things to evaluate. Our observations also include a focused evaluation of the implications of retiring some or all of the existing systems, from many stakeholders’ points of view; this may exclude some strategies. The cost of the different integration strategies is clearly an important influence, the relation to other activities in the organization, how to transfer or support existing data, and how user processes will change. Not only the future system as such must be considered when choosing a strategy, but also the implementation process. The implementation process depends heavily on the strategy, so that the activities needed for Merge will be considerably different from the retirement and evolution activities of Choose One and Start from Scratch. A Merge will involve a long-term synchronization of two parallel systems, and requires stepwise deliveries and some means of achieving momentum in the evolution of the existing systems, in order to make them converge. For many organizations, these aspects are more unknown than to retire some systems and/or evolve or develop a new system, and so it appears as Merge is the most difficult. The trend among our cases is also that Start from Scratch was preferred over Choose One, which was preferred over Merge. There is also the choice of No Integration which has no direct costs, but of course brings no integration benefits – however this may be the best option is the costs and risks of integration is high and the expected benefit is low.

All this taken together suggests that in-house integration is a difficult endeavor, which is also shown by the events in the cases. Reality is always richer and more complex than any systematic description or abstraction. Nevertheless, we believe we have provided a set of concepts that are useful for describing and explaining many of the events in the cases, and also being useful for future in-house integration efforts, to minimize the risk of insufficient analysis and ill-founded decisions.

10.1 Future Work

Since this work can be considered qualitative research to create theory, validating and quantifying these results using more cases is a natural continuation. This is currently done with a questionnaire survey distributed to the same and other cases [60], and we expect this data collection to continue for a while.

We also want to penetrate some of the topics further, and are especially interested in high-level and rapid reasoning about compatibility, i.e. at the architectural level. We intend to research the Merge strategy further and are currently following up case F2 in order to develop a method and a tool [49,56] for supporting rapid exploration of different Merge alternatives and evaluate them.

The available knowledge within an organization would also be an important input to the decisions made deserve further studies; for example, merging and reusing systems are less feasible options if (either of) the systems are not properly documented and the original architects have left the organization.

We also welcome studies of this topic focusing less on technical factors and more on other factors such as management, cultures and psychology, and ethical dilemmas such as how to handle staff in a stressing situation when retiring a system. We believe there is much knowledge to collect in these fields and synthesize with the more technical point of view put forward in this article.
10.2 Acknowledgements

We would like to thank all interviewees and their organizations for sharing their experiences and allowing us to publish them. Thanks also to thank Stig Larsson and Laurens Blankers for all cooperation during the research that has led to this paper.

11 References


Land R., Thilenius P., Larsson S., and Crnkovic I., A Quantitative Survey on Software In-house Integration, MRTC report ISSN xxx-xxx ISRN MDH-MRTC-xxx (before formal publication and assignment of ISSN/ISRN, it is available at: www.idt.mdh.se/~rld/temp/survey_tr.pdf), Mälardalen Real-Time Research Centre, Mälardalen University, 2006.


Appendix: Interview Questions

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.?

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?
Figure 1. The batch sequence architecture of the existing systems in case F2. Arrows denote dependency; data flows in the opposite direction.

Figure 2. The currently common and different parts of the systems in case F2.

Figure 3. The currently common and different parts (exemplified) of the 3D simulators in case F2.
Figure 1. The batch sequence architecture of the existing systems in case F2. Arrows denote dependency; data flows in the opposite direction.
Figure 2. The currently common and different parts of the systems in case F2.
Figure 3. The currently common and different parts (exemplified) of the 3D simulators in case F2.
Table 1: Summary of the cases.

<table>
<thead>
<tr>
<th>Organization</th>
<th>System Domain</th>
<th>Goal</th>
<th>Information Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Newly merged international company</td>
<td>Safety-critical systems with embedded software</td>
<td>New HMI(^1) platform to be used for many products</td>
</tr>
<tr>
<td>B</td>
<td>National corporation with many daughter companies</td>
<td>Administration of stock keeping</td>
<td>Rationalizing two systems within corporation with similar purpose</td>
</tr>
<tr>
<td>C</td>
<td>Newly merged international company</td>
<td>Safety-critical systems with embedded software</td>
<td>Rationalizing two core products into one</td>
</tr>
<tr>
<td>D</td>
<td>Newly merged international company</td>
<td>Off-line management of power distribution systems</td>
<td>Reusing HMI(^1) for Data-Intensive Server</td>
</tr>
<tr>
<td>E(_1)</td>
<td>Cooperation defense research institute and industry</td>
<td>Off-line physics simulation</td>
<td>Creating next generation simulation models from today’s</td>
</tr>
<tr>
<td>E(_2)</td>
<td>Different parts of Swedish defense</td>
<td>Off-line physics simulation</td>
<td>Possible rationalization of three simulation systems with similar purpose</td>
</tr>
<tr>
<td>F(_1)</td>
<td>Newly merged international company</td>
<td>Managing off-line physics simulations</td>
<td>Possible rationalization by using one single system</td>
</tr>
<tr>
<td>F(_2)</td>
<td>Newly merged international company</td>
<td>Off-line physics simulation</td>
<td>Improving the current state at two sites</td>
</tr>
<tr>
<td>F(_3)</td>
<td>Newly merged international company</td>
<td>Software issue reporting</td>
<td>Possible rationalization by using one single system</td>
</tr>
</tbody>
</table>

\(^1\) HMI=Human-Machine Interface
Table 2: The exclusion of possible strategies

<table>
<thead>
<tr>
<th>Architectural Compatibility: “The similarity of existing systems is…”</th>
<th>Start from Scratch</th>
<th>Choose One</th>
<th>Evolutionary</th>
<th>Rapid</th>
</tr>
</thead>
<tbody>
<tr>
<td>“…very high”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“…modest”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“…very small”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Retireability: “It is possible to retire…”</th>
<th>Start from Scratch</th>
<th>Choose One</th>
<th>Evolutionary</th>
<th>Rapid</th>
</tr>
</thead>
<tbody>
<tr>
<td>“…all”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“…some”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“…none”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend: By following the rows corresponding best to the situation in a case (one for architectural compatibility and one for retireability), black denotes what strategies are excluded.
Table 3. The types of reuse for the different artifacts in the cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Requirements</th>
<th>Architectural solutions</th>
<th>Components</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**System level**

**First level of decomposition:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Requirements</th>
<th>Architectural solutions</th>
<th>Components</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>D:HMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D:Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D:GIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2:Pre</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2:2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2:Post</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2:3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Second level of decomposition:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Requirements</th>
<th>Architectural solutions</th>
<th>Components</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>D:2D:Post</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D:2D:Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

- Reuse from all
- Reuse from one
- No reuse
- Unsure classification, see numbered comments below

1. Architectural solutions were reused mainly from one, with heavy influence from one of (several) other systems.
2. It is unknown what will be reused in the future integrated system.
3. One component (the post-processor) started out as an attempt to reuse from System 2, but in the end only a small fraction of the original component was left and should probably be considered source code reuse.
4. It is unsure whether any source code was reused from the retired system (not enough information).
5. The future systems will be an evolution of System 1, while incorporating a methodology from System 2; it is not known whether this means reuse of certain architectural solutions and components (the latter seems unlikely).
Table 4: Summary of the possible and desired strategies in the cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Start from Scratch</th>
<th>Choose One</th>
<th>Evolutionary</th>
<th>Rapid</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (initial)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C (final)</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>*D</td>
<td>(?)</td>
<td>(?)</td>
<td>Y</td>
<td>(?)</td>
</tr>
<tr>
<td>D_HMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*D_Server</td>
<td>(?)</td>
<td>(?)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1 (initial)</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>*F1 (second decision)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2_pre</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2_2D</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>F2_post</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2_3D</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>F3</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend: Each row denotes a case. There are multiple entries for case C and F1, to capture how evaluation and/or decision changed. There are also entries for the constituent components of cases D and F2, where the Merge strategy was chosen and currently implemented. For each case, black denotes the strategies that are practically excluded, based on table 2 and our interpretation of the cases; a question mark indicates that our classification is open for discussion. A check mark indicates which strategy was finally selected and implemented.
Paper III

This paper is a reprint of:

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

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.

Figure 1 describes the architectures of the three existing systems in a high-level diagram blending an execution view with a code view [3]. The most modern system is built with a three-tier architecture in Java 2 Enterprise Edition (J2EE), while the two older systems are designed to run in a Unix environment with only a thin “X” client displaying the user interface (the “thin” client is denoted by a rectangle with zero height in the figure); they are written mostly in Tcl and C++, and C++ with the use of Motif. The Tcl/C++ system contains ~350 KLOC (thousands of lines of code), the C++/Motif system 140 KLOC, and the Java system 90 KLOC. The size of the rectangles in the figure indicates the relative sizes between the components of the systems (as measured in lines of code). The Tcl/C++ system uses a proprietary object-oriented database, implemented as files accessed through library functions, while the two other systems, which were developed at the same site, share data in a common commercial relational database executing as a database server.
Since the two software development departments (the North American and the European) had cooperated only to a small extent beforehand, the natural starting point was simply to meet and discuss solutions. The managers of the software development departments accompanied by a few software developers met for about a week, outlined several high-level alternatives and discussed their implications both in terms of the integrated system’s technical features and the impact on the organization. Since the requirements for the integrated system was basically to provide the same functionality as the existing systems, with the additional benefits of having access to more and consistent data, user involvement at this early stage was considered superfluous. At this meeting, no formal decision was made, but the participants were optimistic afterwards – they had “almost” agreed. To reach an agreement, the same managers accompanied with software developers met again after two months and discussed the same alternatives (with only small variations) and, once again, “almost agreed”. The same procedure was repeated a third time with the same result: the same alternatives were discussed, and no decision on an integrated architecture was made. By now, almost half a year had passed without arriving at a decision.

Higher management insisted on the integration and approved of a more ambitious project with the goal to arrive at a decision. Compared to the previous sets of meetings, it should contain more people and involve more effort, and be divided into three phases: “Evaluation”, “Design”, and “Decision”, with different stakeholders participating in each; see Figure 2. First, the users were supposed to evaluate the existing systems from a functional point of view, and software developers from a technical point of view. Then, this information should be fed into the second phase, where software developers (basically the same as in phase one) should design a few alternatives of the architecture of an integrated system, analyze these, and recommended one. In the last phase, the managers concerned were to decide which architecture to use in the future (maybe, but not necessarily, the one recommended in phase 2). The first phase lasted for two weeks, while the second and third phases lasted for one week each.

Of course, this characterization is somewhat idealized – in reality, there were more informal interactions between the stakeholder groups and between the phases: briefings were held almost each day during the course of the meetings, to monitor progress, adjust the working groups’ focus etc.
Phase 1: Evaluation. Six users experienced with either of the three systems had hands-on tutorials and explored all the existing systems, guided by an expert user. They produced a high-level requirements specification with references to what was good and less good in the existing systems. In general they were content with the existing systems and were explicit in that it was not necessary to make the user interface more homogeneous; they would be able to work in the three existing user interfaces, although very dissimilar. The user evaluation would therefore not affect the choice of architecture.

The developers found that although the existing systems’ documentation included overall system descriptions, they were of an informal and intuitive kind (for example, none of them used UML), which meant that the descriptions were not readily comparable, making the development of architectural alternatives difficult. During the first phase, the developers were therefore to produce high-level descriptions of the existing systems that would be easily comparable and “merge-able”.

Figure 1. Today’s three systems.
Phase 2: Design. In phase 2, the software developers tried several ways of “merging” these architectural descriptions. Their experience and knowledge of the existing systems was the most important asset. Two main alternatives were developed, a “data level” integration (preserving the differences between today’s systems but adapting them to use the same database, see Figure 3a), and the “code level” integration alternative (using the three-tiered architecture of the existing Java system, see Figure 3b). The architectural descriptions were analyzed briefly regarding functionality and extra-functional properties such as performance, maintainability, and portability, and project plans for the implementation of the two alternatives were outlined. The developers recommended the “code level” alternative due to its many perceived advantages: it would be simpler to maintain, bring the users more value, be perceived by users as a homogeneous system, while not being more expensive in terms of effort to implement (according to the estimations, that is).

Figure 2. Project phases.
Phase 3: Decision. All written documentation (architectural descriptions, project plans for their implementation, and other analyses) was forwarded to the third phase. The managers concerned had a meeting for about a week when they discussed costs, risks, business implications, organizational impact, etc. of the two alternatives. It was decided that the systems should be integrated according to the “data level” alternative, since this solution was considered to be associated with a lower risk than the “code level” alternative; risk meaning the probability of overrunning budget and/or schedule, producing a product of poor quality, or fail altogether with the integration. The risk parameters are not only those related to technical problems (such as those involved with writing new code), but also the risk of successful collaboration (in terms of “commitment required” from departments of two previously separate organizations, not yet so close collaborators).

3. Analysis

While a handful of alternatives were discussed during the first meetings, there were only two alternatives produced in the design phase of the three-phase project. The alternatives themselves were not new – the developers almost indignantly said that they discussed the same alternatives and issues as they had done for six months. It was rather the ability to agree on
discarding some alternatives with a certain amount of confidence that was an improvement as compared to the first sets of meetings. Assuming that the developers were correct in that the discarded alternatives were inferior, this reduction of the numbers of alternatives was arguably an improvement compared to the first sets of meetings. The managers in the third phase had “only” to choose between these two alternatives, and as we described, the users did not favor any of these, which made it possible for the managers to base the decision on a smaller set of concerns.

In the rest of this section, the features of the process that enabled these improvements are discussed. We highlight what we believe to be good practices in general during software integration as well as challenges for the future. These conclusions are partly based on a questionnaire responded to by (some of) the participants of the projects.

**Early meetings.** In a newly merged organization, the “people aspect” of software integration needs to be addressed, and meeting in person to discuss integration in general, and even particular alternatives, is the most important means to build the trust and confidence needed. This should not be seen as a replacement for a more structured project, however.

**Several-phase process.** By dividing the stakeholders into different activities with specific tasks, the discussions become more focused and efficient. At the same time, more interaction that only forwarding deliverables is needed; in the project, briefings were held almost every day involving people concerned, to monitor progress and adjust focus if needed. The scheme used does not differ from already documented good practices in other software activities, such as development and maintenance.

**User involvement.** Performing a user evaluation of existing systems prior to integration is crucial. If the outcome does not affect the choice of architecture, this is good news for the decision process – the choice can be made based on other concerns. Moreover, any issues found during the user evaluation are important inputs to subsequent phases, during actual implementation. Since the user evaluation did not affect the choice in the case study however, it did not really fulfill the developers’ expectations. We therefore suggest that in an integration process the expectations should be clearly articulated. If the goal of the user involvement at this early stage is to assess whether they have any preferences that affects the choice of architecture, the type of evaluation performed in the case study seems reasonable – enough users must be given time to understand the systems in enough depth to achieve a certain amount of confidence in the analysis results. However, if the goal is to take the opportunity of improving the
existing systems significantly when integrating them, the situation reminds of development of new software, and established requirements engineering, more heavily involving users and other stakeholders, should then be applied [4]. The existing systems can be thought of as a requirement specification or prototype in evolutionary or spiral development [1]. A cheap, initial investigation involving users may indicate that a more thorough evaluation is needed.

Separating Stakeholders. This should be no surprise – it does not make sense to bring all stakeholders together for all meetings during the process. We have showed a three-phase process where the separation of stakeholders made the meetings more efficient and focused. The discussions were kept at a level detailed and technical enough to enable fruitful discussions since the participants had similar background and roles. By assigning different tasks to the different phases, the responsibilities became clearer. The developers could first concentrate on evaluating the existing systems, and only later bother about their integration. The managers were reduced to “only” making a decision, basically by choosing between two alternatives with certain properties.

Active upper management. Upper management insisted that the systems should be integrated: implicitly, since they once again started a project with the same goal, and more explicitly by deciding on a date when there had to be a decision. There was an integration coordinator, responsible for all integration activities resulting from the company merger, who actively showed interest in the project.

Architecture-centric process. During many software activities, the process can benefit from being oriented around the architecture of the system being built [8]. How the architecture was used in this particular case study has been described in more detail elsewhere [5,6].

Different people. Although there were developers and managers participating in each project execution the people participating in each meeting or in the final project were not identical. Perhaps the mix of people in the successful project was a successful blend of open minds, while in the previous meetings this was not the case? According to the questionnaire data, this might be the case.

It will take time. Eight months passed from the initial meetings to the decision. This means that the project members and the managers had got to know each other better on a personal level, and overcome cultural differences between the two countries and formerly separate organizations [2]. When a decision is dependent on people collaborating for the first time,
especially when they have different cultural backgrounds (as is the case after mergers, especially international ones), it must be expected that the process will take more time than a project executed completely within either of the departments – and possibly also a higher amount of disagreement and frustration. With this in mind, it is likely that the actual integration also will take time, and that an integration project in the context of a company merger will face more obstacles in terms of cultural differences and priority clashes than a project within either of two collaborating departments would do.

4. Summary

After a company merger, an organization typically wants to integrate its software tools. In this paper, we investigated a case study illustrating how this can be done, and pointed out some key features of such a process that can be summarized as early meetings, several-phase process, user involvement, separating stakeholders, active upper management, architecture-centric process, different people, and not least: it will take time.

5. References


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]:

- The influence of existing systems’ architecture on what can be reused into a future system.
- Beneficial process practices and risk mitigation tactics, based on the interviewees’ own descriptions of mistakes and successes.
- Available high-level strategies and the most important factors to exclude some of them.

The present paper presents a study in which a questionnaire was designed and distributed in order to validate and quantify the earlier observations. Section 2 describes the research method employed. The questionnaire and the results are presented in Section 3. Section 4 relates this work to existing research, and Section 5 concludes the paper.
2. Research Method

Based on the results of the previously mentioned qualitative and explorative study, we constructed a questionnaire which was administered to people that had participated in in-house integration efforts. The questionnaire was constructed with some one hundred statements concerning their project, divided into four groups connected to our earlier findings: high-level strategies and decision-making, reuse and retirement, technologies and architectures of the existing systems, and process practices. All statements were to be graded with the same scale, ranging from 1 (“I do not agree at all”) to 5 (“I agree completely”).

2.1 Purposes

In relation to the previous multiple case study, there are three purposes of the questionnaire:

- **Validation.** By returning to the previous cases, some amount of internal validation of our previous interpretations (in terms of theory construction) is ensured. If the same cases are described in a very different way from our previous interpretations based on interviews, it is a sign that the theory is a bad representation of the reality. Also, 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.

- **Quantification.** Given that internal validity is satisfactory, some of the previous qualitative observations can be quantified. For example, the previous study led to a list of suggested beneficial practices, which can now be ranked in importance.

- **Hypothesis testing.** Our previous analyses includes some suggested correlations and causal links which can now be tested. This means that we search for uniform patterns across the respondents if the answers to two (or more) questions vary together.

2.2 Sampling Method and Response Rate

From a statistical point of view, the ideal situation would be to define the population (all in-house integration projects) and then sample randomly from
it. However, although there exist databases with e.g. all companies in Sweden, there is no known way of find all newly merged companies. It would also be problematic and time-consuming to find the right persons within the selected companies, and there is no inherent reason to exclude non-commercial software developing organizations, or foreign organizations. The most resource-efficient method was to use convenience sampling, i.e. we identified potential projects and respondents through personal contacts.

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. The questionnaire and the full nine responses are published as a technical report [17]. The cases come from various domains, including safety-critical software, calculation-intensive simulation programs, and data-intensive systems. We make no analyses here concerning possible differences across domains or sizes of programs or organizations but refer to [17] for anyone who wants to draw their own conclusions. The naming of the cases is chosen as to be consistent with that of previous publications [16].

3. Results

This section is organized per question studied. Each subsection describes 1) the previous observations, 2) a summary of the questions in the questionnaire aimed at validating or quantifying these previous observations, 3) the most important questionnaire results, and 4) some interpretations of these results.

3.1 Integration Strategies

Based on the qualitative data from the earlier multiple case study, four high-level integration strategies have previously been described:
• **Start from Scratch** Discontinue all existing systems and initiate the implementation of a new system. The new system will likely inherit requirements and architecture from the existing systems.

• **Choose One** Evaluate the existing systems, choose the one that is most satisfactory, evolve it if necessary, and discontinue all others.

• **Merge** Take parts from the existing systems and integrate them to form a new system that has the strengths of both.

• **No Integration** Do nothing (mentioned to be complete).

The questionnaire contains a number of questions intended to identify the selected strategies in the cases. The purpose is twofold: first, the questionnaire results would indicate how well cases in reality fit within the proposed classification. Second, for the cases of the previous multiple case study, the questionnaire answers would reveal how well our earlier interpretations (in terms of selected strategy) match these more directed questions, i.e. the questionnaire can validate our previous interpretations.

### 3.1.1 Validation

Profiles that describe a set of responses that would match a specific have been prepared. For example, the statement “All existing systems is (or will be) retired” should for the *Choose One* strategy yield the response 1 (“I do not agree at all”), while for *Start from Scratch* the response should be 5 (“I agree completely”). Each case was then matched with the profiles. This was done by calculating the absolute difference between the actual response and the strategy profile for each question, and calculating the sum for all questions.

<table>
<thead>
<tr>
<th>Case</th>
<th>Earlier Interpretation</th>
<th>Questionnaire Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td><em>Start from Scratch</em></td>
<td><em>Start from Scratch</em></td>
</tr>
<tr>
<td>B</td>
<td><em>Choose One</em></td>
<td><em>Start from Scratch</em></td>
</tr>
<tr>
<td>C</td>
<td>Changed decision from <em>Merge</em> to <em>Choose One</em> (which was implemented, with some reuse)</td>
<td>Impossible to single out one clear strategy</td>
</tr>
<tr>
<td>E1</td>
<td><em>Start from Scratch</em></td>
<td><em>Start from Scratch</em></td>
</tr>
<tr>
<td>E2</td>
<td><em>Choose One</em>, but with some reuse (i.e. resembling <em>Merge</em>)</td>
<td>Difficulty to distinguish between <em>Choose One</em> and <em>Merge</em></td>
</tr>
</tbody>
</table>

**Table 1: Comparison of earlier and current interpretations.**
Five of the previous nine cases were revisited, and the previous interpretations and the questionnaire results were compared. The results are shown in Table 1. For cases A and E1 the results are in harmony with each other. For case C, E2, and F2 the questionnaire results do not clearly point at one single strategy, but not conflicting with previous interpretations. (Actually, our earlier interpretation of case C was that the vision was unclear and direction changed several times, which the questionnaire data might reflect; the respondent also made several notes in the margin of the questionnaire like “depends on what point in time”. For case F2, which we interpreted as a Merge, there are actually several separate sub-systems that were earlier interpreted as Choose One and Start from Scratch; it is unclear exactly which system or subsystem the respondent had in mind when answering the questionnaire.) For case B, which we earlier considered a clear case of Choose One, the result is now a clear Start from Scratch. Returning to this particular respondent’s questionnaire data, some of the questionnaire responses are exactly contrary to what was said during earlier interviews; we find no other interpretation than that some of the questions must have been ambiguous and misinterpreted (we believe the qualitative data closer to the truth).

Our conclusions must be: the questionnaire data are primarily in line with our earlier interpretations. However, many cases in reality would be described as hybrids (e.g. Choose One with some reuse). The strategies might still be useful as a model in discussions and planning. Also, the contradictory data given in cases B and G calls for refinement of the questionnaire before it is used in further research.

3.1.2 Decision Making Considerations
A number of questions were asked concerning how the high-level decision was made. The responses to each question are summed, and scaled to the percentage of the possible maximum (i.e. if all responses were “5”). The
result is shown as a ranked list of statements in Table 2. The statements that scored highest are the ones with the most homogeneous answers, while for others opinions tend to differ more among the respondents.

**Table 2: The relative importance of decision making considerations.**

<table>
<thead>
<tr>
<th>The high-level decision about how to integrate…</th>
<th>Importance (Percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...was made by management</td>
<td>84</td>
</tr>
<tr>
<td>...was based on technical considerations</td>
<td>71</td>
</tr>
<tr>
<td>...was based on considerations concerning the parallel maintenance and evolution of existing systems</td>
<td>63</td>
</tr>
<tr>
<td>...was made by technicians</td>
<td>62</td>
</tr>
<tr>
<td>...was based on considerations for existing users</td>
<td>60</td>
</tr>
<tr>
<td>...was based on available staff and skills</td>
<td>60</td>
</tr>
<tr>
<td>...was based on politics</td>
<td>60</td>
</tr>
<tr>
<td>...was based on considerations on time schedule</td>
<td>38</td>
</tr>
</tbody>
</table>

We see that management makes the decision rather than technicians, although they are also involved. The most important bases for decision seem to be of technical nature and concerns about the organization. Considerations on time schedule seem to be the least important consideration in in-house integration, which might indicate that such strategic tasks must be allowed to take the time and resources it requires; there is perhaps no perceived alternative. (Here we should note that any possible No Integration cases were probably systematically excluded from participation in the survey, as we initially searched for cases where integration had actually happened, or was underway.) The distribution of answers was generally larger for statements with lower importance.

### 3.2 Retirement of Existing Systems

We previously found the discussions concerning the impact of retiring the existing systems to be an important influence when selecting an integration strategy, and the questionnaire was designed to elaborate this somewhat. The exactly same statements used when comparing with the strategy profiles were to be graded both according to what was management’s vision for the integration, and what was the actual outcome (or seem to be, if it is not finished yet). The largest differences between vision and eventual result
concerned retirement of systems, but interestingly the differences were in both directions: sometimes management wished to retire but this never happened, and sometimes management did not plan to retire but eventually some system(s) were retired nevertheless. The lesson learned is that one needs to pay extra attention to the issue of retiring systems.

The respondents were asked whose opinions the decision on retireability was based on, among customers, users, developers, marketing people, and management. Only small differences were found (with fairly large distribution), so the conclusion is that all of these seem to have been involved to the same extent.

There were also a set of questions concerning backward compatibility of the future system, i.e. whether the future system: 1) needs to support the way users currently work, 2) be backwards compatible with existing data, 3) be backwards compatible with existing surrounding tools, and 4) be backwards compatible with installations of the existing systems. No obvious difference was found; if anything, the last aspect seems to be the least important. The differences were large between the cases though: some cases scored high for all these four aspects, others low. This means that other factors such as market situation probably have a significant influence on these requirements, and that we based on our investigations are unable to formulate general guidelines.

### 3.3 Architectural Compatibility

We previously found the architectures of the existing systems to be important when selecting an integration strategy, and as described in section 3.1.2 technical considerations seem to be influential when deciding on a strategy. The respondents were asked to grade how similar the existing systems were according to a number of criteria.

The differences were large between the cases, and when summed across the cases (and scaled to the percentage of the possible maximum), clear differences can be discerned as to how common some particular similarities were in general. See Table 3. The questionnaire was designed so that each such criterion was considered part of either structure (S), data model (DM), or technology/framework (TF). When comparing these groups, no general differences can be identified.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Similarity</th>
</tr>
</thead>
</table>

Table 3: Similarities among the systems in the cases.
The single similarity that stood out as the most common (with small distribution of answers) was that the existing systems contain software parts/components/modules with similar functionality. This should not be too surprising, since the systems in each case are in the same domain, solving similar problems. Both systems to be integrated should contain a user interface component, a physics calculation component etc. if they are both built to address a problem that requires a user interface and physics calculations. This can be compared with the statement “the software of the existing systems have the same internal structure (architecture)”, which scored lowest of all criteria. This would mean that although there are components with similar roles in both systems (e.g. user interface, physics calculations, etc.), these are organized in different ways. Also scoring very low is “the parts/components/modules exchange data in the same ways in the
existing systems”. Together, this suggests that picking the best components from different systems and reassembling them into a new system (the Merge strategy) is most likely very difficult. This is in line with the well-known observation on “architectural mismatch” [9].

We also analyzed the connection between the three groups of similarities and three possible sources of similarities that we have previously identified. This is done by plotting the similarities and the responses in each case concerning possible sources of similarities (exemplified by Figure 1). Based on the current data, if the systems are initially built in the same time period, the technology/framework and data models there seem to be a tendency of similarity (see Figure 1). If the systems implement the same domain standards, such as for safety-critical applications, this seems to be linked to all types of similarities. One previously reported cause of similarities is when systems have been branched from the same system as a result of earlier collaborations. However, based on the questionnaire data it is impossible to claim such a link, which is somewhat surprising. The respondents might have interpreted “common ancestry” in the questionnaire differently than was our intention, and this needs to be considered in further research.
The existing systems were initially built in the same time period (e.g. decade).

Figure 1: Indications of a link between same time of origin and similarities.

3.4 Exclusion of Strategies?

In earlier research we have presented the hypothesis that a certain amount of compatibility or similarity between the existing systems is a prerequisite for Merge, and that to be able to choose the strategies Start from Scratch and Choose One it must be considered feasible to retire (some of) the existing systems. The questionnaire was designed to be able to correlate the strategy questions with the compatibility questions and the retirement questions, but the low number of respondents and the weak support for describing some cases with a single strategy makes it difficult to draw any definite conclusions. Table 4 presents the cases in decreasing order of similarity, and we see that the cases that mostly resemble a Merge (as described in section 3.1.1) are at the top.

Table 4: The similarities in the cases.

<table>
<thead>
<tr>
<th>Case</th>
<th>Strategy Selected</th>
<th>Similarity: Average (S/DM/TF) (Percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>Merge / Choose One</td>
<td>76 (77/80/70)</td>
</tr>
<tr>
<td>E2</td>
<td>Merge / Choose One</td>
<td>70 (50/80/80)</td>
</tr>
<tr>
<td>A</td>
<td>Start from Scratch</td>
<td>64 (63/60/70)</td>
</tr>
</tbody>
</table>
Similar questions, but fewer, were asked in order to investigate how discussions and decisions on retireability are linked to the choice of strategy. No clear difference can be found between the cases on how statements on retirement have been graded. Retirement will arguably depend on the situation on the market, and we still believe that retirement and choice of strategy must be discussed together. This means that the question is complex and the measurement instrument must be refined; it is probably necessary to more clearly distinguish between the message to the market and the plans for the actual implementations. For example, a completely new implementation could be marketed under an old, popular name.

3.5 Process Practices

Based on the previous multiple case study data, the questionnaire listed a number of process practices and risk mitigation tactics. Compared to previous publications, these statements were somewhat shortened (and in some cases divided into several statements) to make them more straightforward to rank. The respondents were asked to grade: 1) how important this was for project success, and 2) how much attention it was given in the project. Both these grades were summed across the cases (and scaled to the percentage of the possible maximum). In Table 5, the practices are enumerated in decreasing order of importance. The attention given is also listed, as well as the difference between importance and attention.

The highest score, with the smallest spread among answers concerns the need of management’s commitment, shown by allocating enough resources. The responses also show that a strong project management is needed, and that the grassroots – i.e. the people who will actually do the hard and basic work – must be cooperative, both with management and each other. Another practice that scored high was that a small group of experts should be assigned early to evaluate the existing systems and describe alternative high-level strategies for the integration. A number of practices scored somewhere in between, and the two that scored lowest in total (although with greatest distribution of answers) were that the future system must contain more features than the existing systems (which one would expect in order to
sustain commitment for a long and expensive project), and that the future system should be described in terms of the existing systems (which we previously found to be an intuitive way of working rapidly with requirements). That they scored lowest does not mean they are unnecessary, only considered less important when compared with other practices.

Even more interesting than studying perceived importance on its own, is to compare it with how much attention the practices received in the projects, as an indication of how often these practices are neglected. The greatest difference between importance and attention was found among the practices graded as most important. Ranked by this difference, together with the statements on commitment (from both management and grassroots) and the need for a strong project management, we find the need for more formal agreements between sites than usual. The fact that these four practices have not gained enough attention illustrate the challenges involved in coordinating two previously separate organizations with their existing local priorities, as well as the well-known challenges of distributed software development [5,6,11].

Table 5: Process practices.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Importance (Percent)</th>
<th>Attention</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management needs to show its commitment by allocating enough resources</td>
<td>98</td>
<td>58</td>
<td>40</td>
</tr>
<tr>
<td>A strong project management is needed</td>
<td>95</td>
<td>68</td>
<td>28</td>
</tr>
<tr>
<td>The “grassroots” (i.e. the people who will actually do the hard and basic work) must be cooperative, both with management and each other</td>
<td>93</td>
<td>78</td>
<td>16</td>
</tr>
<tr>
<td>A small group of experts must be assigned early to evaluate the existing systems and describe alternative high-level strategies for the integration.</td>
<td>93</td>
<td>74</td>
<td>18</td>
</tr>
<tr>
<td>Experience of the existing systems from many points of view must be collected.</td>
<td>91</td>
<td>73</td>
<td>19</td>
</tr>
<tr>
<td>All stakeholders must be committed to the integration</td>
<td>89</td>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td>A common development environment is needed</td>
<td>82</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>Decisions should wait until there is enough basis for making a decision</td>
<td>80</td>
<td>58</td>
<td>22</td>
</tr>
</tbody>
</table>
4. Related Work

The insight that software evolves, and has to evolve, is not new \[18,21\]. Software has to be extended in different ways to keep up with evolving needs and expectations, and interoperability and integration is one type of extension. There is literature describing integration rather fundamentally in terms of a) interfaces and interoperability \[27,28\], b) architecture \[1,9,12\] and architectural patterns \[4,23\], and c) information/taxonomies/data models \[10\]. These foundations are directly applicable to the context of in-house integration.

The concept of integration is fundamental in the field of Component-Based Software Engineering \[25,26\]. Also, the notion of standard interfaces and open systems \[20\], builds upon the idea of integration in the sense interoperability. However, these existing fields address somewhat different problems than in-house integration. Integration in these fields means that components or systems complement each other and are assembled into a larger system, while we consider systems with overlapping functionality. The challenge in in-house integration is therefore not to assemble components into one whole, but to take two (or more) whole systems and create one single whole, containing the best of the previous systems. Also, these fields typically assume that components (or systems) are acquired from third parties controlling their development, meaning that modifying them is not an option. We focus on systems completely controlled in-house, and this constraint consequently does not apply.

Enterprise Application Integration (EAI) is a large business as well as a research area \[8,13\]. EAI is a systematic approach to integrating systems acquired from third parties, typically with technologies such as middleware.
and message buses. This is suitable since EAI focuses on data-centered systems, with high requirements on e.g. throughput and scalability. The systems subject to in-house integration could be other types of systems, such as control systems with resource constraints and requirements on safety and real-time behavior, which means that the EAI solutions do not apply. While EAI is a way to interconnect one specific set of installations together, in-house integration concerns integrating systems that may be deployed and installed millions of times. The focus shifts towards efficiency per installation, both in terms of man-time and runtime.

There are established software development models, ranging from the traditional sequential waterfall model with different variants [19], iterative, incremental, and evolutionary models [3,19], the commercially marketed Rational Unified Process (RUP) [15] and agile methodologies [2,24]. The focus is usually on new development, although there is also literature on component-based development processes [7,26]. It is already known that reusing software is difficult [14,22], something confirmed also in the case of in-house integration. There is a body of knowledge on global and distributed software processes, also focusing mostly on new development [5,11]. Many identified challenges seem to hold also for in-house integration, e.g. overcoming physical distance and cultural differences [6].

5. Summary and Conclusions

This paper presented a study of in-house integration projects that validates and quantifies earlier observations. The low number of respondents means that interpretations must be very carefully made. Relating the current study with earlier research, the method that has been validated, and thus serves as suggestion for future projects, can be described as follows.

The notion of high-level strategies might be useful as a model during discussions, in that they represent the extreme ends of the solution space. However, no case is in reality easily described with one single strategy. This reflects that reality is more complex than models of it. Strategy selection is usually made by management rather than technicians, but is based on technical considerations. This might indicate that there is a higher awareness among managers of the potential technical problems than we have reported earlier. Based on the data, the least important consideration when selecting a strategy is the cost and time associated with the integration. This might indicate the strategic importance of integration. The option of not integrating is often not considered a real option, so the integration must be allowed to
take time. In this part of the study, there were a few instances of responses that are contrary to earlier interviews with the same respondents, which call for further investigation and improvement of the measurement instrument.

We have found some support for the hypothesis that Merge is only possible if there is a certain amount of similarities. Three types of similarities are presented that should be investigated in future projects: structure, data model, and technology/framework. Across our cases, these types of similarities were roughly equally common, but within a case the types of similarities found differed greatly. The greatest difference in general is found between two measures of structure similarities: although it is common that there are components with the same roles in systems addressing the same problem in the same domain, the structure of these components is usually different (contrary to our earlier observations). The data indicates a connection between the time period when a system was built and similarities; some similarities can also be expected if there are domain standards implemented by the systems.

To achieve the necessary combination of technical knowledge and timely decision of a high-level strategy, a small group of technical experts should be gathered early; this practice was reported as important across the cases. Such a group would evaluate the existing systems and outline and analyze possible (and impossible) future, integrated systems. Other practices reported as important concern commitment of stakeholders – in management’s case this must be accompanied with providing adequate resources. One important lesson is that there is a large difference for these and more practices, between the perceived importance and to what extent these are actually employed in reality. In particular, the challenge of making two previously separate organizations strive towards the same goal is often underestimated; one solution is to have strong management, make agreements between sites more formally than either site is normally used to in in-house projects, and to have a mechanism for enforcing these.

For future software in-house integration projects, there are several lessons to be learned. First, there are a number of important practices that are often overlooked; future projects should therefore ensure that these practices are employed. Second, approaches with much reuse from several systems (expressed as a Merge in the extreme case) must have a certain degree of similarity to succeed. Third, even if many of these results make intuitive sense, it might be a reassurance for future decision-makers to know that others have successfully made similar decisions.
5.1 Future Work

Our intention is to provide a refined analysis based on a larger number of cases. That would also make it possible to distinguish possible differences between e.g. application domains and sizes of organizations. We would also like to take part in projects where some of the current observations and suggestions are systematically employed and evaluated.

5.2 Acknowledgements

We would like to thank all respondents and their organizations for sharing their experiences and allowing us to publish them. Thanks to Laurens Blankers at Eindhoven University of Technology for previous collaborations in this research field. Thanks to Cecilia Erixon at School of Business at Mälardalen University for help with designing the questionnaire.

6. References


Paper V

This paper is a reprint of:

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 \( \alpha \) is said to use module \( \beta \) if module \( \alpha \) relies on the correct behavior of \( \beta \) to accomplish its task [14].
In our method, the term \textit{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 \textit{module instance}. In the example, both systems have a \texttt{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 \texttt{EventView}_A and \texttt{EventView}_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., \texttt{EventView\_new\_impl}), an already existing module to be reused in-house from some other program (e.g., \texttt{EventView\_pgm\_name}), or an open source or commercial component (\texttt{EventView\_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 \texttt{systems}, (named with capital letters, and parameterized by \(X, Y, \) etc.).

- A \texttt{module} represents a conceptual system part with a specific purpose (e.g., \texttt{EventView} in Figure 1). Modules are designated with capital first letter; in the general case we use Greek letters \(\alpha\) and \(\beta\).

- A \texttt{module instance} represents a realization of a module. It is denoted \(\alpha_X\) where \(\alpha\) is a module and \(X\) is either an existing system (as in \texttt{EventView}_A) or an indication that the module is new to the systems (as in \texttt{EventView\_pgm\_name} or \texttt{EventView\_component\_name}).

- A \textit{"use" dependency} (or \textit{dependency for short}) from module instance \(\alpha_X\) to module instance \(\beta_Y\) means that \(\alpha_X\) relies on the correct behavior of \(\beta_Y\) 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 $\text{NoteView}_A \rightarrow \text{MusicModel}_A$ and $\text{MouseCtrl}_B \rightarrow \text{MusicModel}_B$.

An adaptation describes that a modification is made to $\alpha_X$ in order for it to be compatible, or consistent with $\beta_Y$, 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: $\langle \text{KbdCtrl}_{\text{new}}, \text{MusicModel}_A \rangle$ $\langle \text{MusicModel}_B, \text{MouseCtrl}_A \rangle$

2.2.2 Inconsistency

A dependency from $\alpha_X$ to $\beta_Y$ can be inconsistent, meaning that $\beta_Y$ cannot be used by $\alpha_X$. 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 $\text{NoteView}_B$ to $\text{MusicModel}_A$ (in System A) and from $\text{EventView}_A$ to $\text{MusicModel}_B$ (in System B). The dependencies from $\text{KbdCtrl}_{\text{new}}$ to $\text{MusicModel}_A$ (in System A) and from $\text{MouseCtrl}_A$ to $\text{MusicModel}_B$ (in System B) on the other hand are consistent, since there are adaptations $\langle \text{KbdCtrl}_{\text{new}}, \text{MusicModel}_A \rangle$ and $\langle \text{MusicModel}_B, \text{MouseCtrl}_A \rangle$ representing that $\text{KbdCtrl}_{\text{new}}$ and $\text{MusicModel}_B$ have been modified to be consistent with $\text{MusicModel}_A$ and $\text{MouseCtrl}_A$ 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 $\alpha_X$ to one of the dependency graphs, if there exists an $\alpha_Y$ in the graph. Additionally, for each module $\beta$, 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 $\beta_W$ in the graph.
4. Add the dependency $\alpha_X \rightarrow \beta_W$ if there exists 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 $\alpha_X$ from one of the dependency graphs, if there are no edges to $\alpha_X$ in the graph, and if the graph contains another module instance $\alpha_Y$ (i.e., with $X \neq Y$).

Note that these operations never change the participating modules of the graphs (if there is an $\alpha_X$ in the initial systems, they will always contain some $\alpha_Y$). 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., $\alpha_X$ because it has richer functionality than $\alpha_Y$, or a new implementation $\alpha_{new}$ because there have been quality problems with the existing $\alpha_X$ and $\alpha_Y$). For other modules, $\alpha_X$ might be preferred over $\alpha_Y$, 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

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 $\alpha_X \to \beta_Y$ 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 NoteViewB and MusicModelA in System A can be solved by adding either of the adaptations $\langle \text{NoteView}_A, \text{MusicModel}_B \rangle$ or $\langle \text{MusicModel}_A, \text{NoteView}_B \rangle$ 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 $\alpha_X$ is exchanged either for $\alpha_Y$ (which is consistent with $\beta_Y$ as they come from the same system $Y$) or for some other $\alpha_Z$ for which there is an adaptation $\langle \alpha_Z, \beta_Y \rangle$ or $\langle \beta_Y, \alpha_Z \rangle$. Alternatively, $\beta_Y$ is exchanged for $\beta_X$ or some other $\beta_Z$ 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, $\text{MusicModel}_A$ could be replaced by $\text{MusicModel}_B$ to resolve the inconsistent dependency $\text{NoteView}_B \to \text{MusicModel}_A$ 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 acquisition 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 “EventView_A is preferred over 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\textsubscript{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 (FMSE and FMUS). The US system chose to incorporate the Swedish module for this, which has required some changes to the modules using the file handling module.
Figure 2: The current status of the systems of the case.

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.

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 (PXSE and PXnew 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 retain 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 PXSE and PXnew 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


Paper VI

This paper is a reprint of:

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).

Figure 1: Initial systems state.
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.

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.

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.
Figure 4: The History View.

Figure 5: The Status View.

Current System A is RED.
The system has incompatible modules linked to its nodes.

Current System B is YELLOW.
The system works, but has nodes or modules which are not preferred.
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).

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


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 222). 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 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”.)

<table>
<thead>
<tr>
<th>Meeting(s) on: dates</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project phase 1 (users)</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project phase 1 (developers)</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Project phase 2</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Project phase 3</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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”.)

<table>
<thead>
<tr>
<th>X</th>
<th>x</th>
<th>X</th>
<th>This made the users’ evaluation more efficient and focused (phase 1).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
This made the developers’ evaluation more efficient and focused (phase 1)

This made the developers’ design and analysis discussions more efficient and focused (phase 2)

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”.)

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>In the first sets of meetings, the responsibilities were unclear, which was one reason that we could not agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td>In the first sets of meetings, the responsibilities were unclear, but this was not significant enough to affect the outcome</td>
</tr>
</tbody>
</table>

In the first sets of meetings, the responsibilities were clear

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”.)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>x</td>
<td>Another meeting similar to the first ones had succeeded in making a decision, if held in October/November</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Another meeting similar to the first ones had not succeeded in making a decision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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”.)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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).
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.

<table>
<thead>
<tr>
<th></th>
<th>One single architect could have been assigned the task of developing alternatives and decide which to use (no separate decision needed)</th>
<th>One single architect could have made the design, and one single manager could have decided</th>
<th>One single architect could have made the design, and several managers could have agreed on a decision</th>
<th>None of the above, because: <em>We needed the knowledge of the designers/users involved. There are no designers/users that has knowledge of all existing systems.</em></th>
<th>None of the above, because: <em>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.</em></th>
<th>Other constellations: <em>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.</em></th>
<th>None of the above, because: <em>there was not one single architect that knew enough about all of the available systems</em></th>
</tr>
</thead>
</table>
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”.)

<table>
<thead>
<tr>
<th></th>
<th>The architectural design solution decided upon was the best</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>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.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>x x x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
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 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”.)
I did not prepare myself enough in advance of the project phases I participated in

I had wanted to spend more time on the project

During the project, I had other more important work to do

I hope I will not be more involved

I gave it my best.

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 didn’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 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 you choose the non-recommended solution? (Please mark zero or more statement with “X”.)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>It means a lower degree of commitment</td>
</tr>
<tr>
<td>x</td>
<td>I would have preferred the recommended solution, but had to compromise</td>
</tr>
<tr>
<td></td>
<td>The project plan the developers produced for the recommended solution was not realistic</td>
</tr>
<tr>
<td></td>
<td>The developers seemed to recommend the solution they did because it was more “elegant” but no more useful</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Comments:

2. Why do you think the other managers chose the non-recommended solution? (Please mark zero or more statement with “X”.)
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.

They thought it would mean a higher risk to rewrite code in a new language.

They thought it would mean a higher cost to rewrite code in a new language.

The level of commitment over time required to complete the plan was not likely to be found within the organization.

---

<table>
<thead>
<tr>
<th></th>
<th>Comments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>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”.)</td>
</tr>
<tr>
<td></td>
<td>To give the user interface a homogeneous look-and-feel</td>
</tr>
<tr>
<td></td>
<td>Because the same functionality would otherwise have to be duplicated in several languages</td>
</tr>
<tr>
<td></td>
<td>Because that would simplify integration of components</td>
</tr>
<tr>
<td></td>
<td>Because some of the languages and technologies used today are very old-fashioned and this will give rise to problems in the future</td>
</tr>
<tr>
<td></td>
<td>Because some of the languages and technologies used today are very old-fashioned and the solution would be less “elegant”</td>
</tr>
<tr>
<td></td>
<td>To simplify maintenance</td>
</tr>
<tr>
<td></td>
<td>To simplify cooperation between Sweden and the US</td>
</tr>
<tr>
<td></td>
<td>To easier attract and keep staff</td>
</tr>
<tr>
<td></td>
<td>To enable the application to be run via the Internet</td>
</tr>
</tbody>
</table>
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

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”.)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>This remark is irrelevant for a decision</td>
<td></td>
</tr>
<tr>
<td>The decision must be based on quantified cost savings, not vague remarks like this</td>
<td></td>
</tr>
<tr>
<td>X This was an important remark which I considered seriously</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

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”.)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>I do not know that we did</td>
<td></td>
</tr>
<tr>
<td>X Some costs were associated with activities we can do without</td>
<td></td>
</tr>
</tbody>
</table>
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”.)

<table>
<thead>
<tr>
<th>No one thought of it</th>
</tr>
</thead>
<tbody>
<tr>
<td>It was too expensive</td>
</tr>
<tr>
<td>A meeting such as the ones that were held was believed to succeed</td>
</tr>
<tr>
<td>It was considered better to mix users, developers, and managers in one meeting than separating them</td>
</tr>
<tr>
<td>x I don’t understand what you are asking</td>
</tr>
</tbody>
</table>

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”.)

<table>
<thead>
<tr>
<th>I do not understand what why it should be unfortunate</th>
</tr>
</thead>
<tbody>
<tr>
<td>The developers’ opinions are not relevant – a company is not a democracy</td>
</tr>
<tr>
<td>x Yes, this may be a source of conflicts, but the chosen solution was so much better so it is worth this risk</td>
</tr>
<tr>
<td>Yes, this may be a source of conflicts, but it was the only way we could agree</td>
</tr>
</tbody>
</table>
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 runtime 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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project Name</td>
</tr>
<tr>
<td>2</td>
<td>My experience in software development activities</td>
</tr>
<tr>
<td>3</td>
<td>My experience with any of the existing systems</td>
</tr>
</tbody>
</table>

Please mark your role(s) in the current project with “X”.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(Technical) architect</td>
</tr>
<tr>
<td>5</td>
<td>Designer</td>
</tr>
<tr>
<td>6</td>
<td>Implementer</td>
</tr>
<tr>
<td>7</td>
<td>Tester</td>
</tr>
<tr>
<td>8</td>
<td>Project leader</td>
</tr>
<tr>
<td>9</td>
<td>Line manager</td>
</tr>
<tr>
<td>10</td>
<td>Product responsible/owner</td>
</tr>
<tr>
<td>11</td>
<td>Other</td>
</tr>
<tr>
<td>Comments</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
</tbody>
</table>


**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”.*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>The existing systems will continue to be maintained, evolved and deployed completely separately.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>One of the existing systems is (or will be) evolved into a common system.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>One or more systems has been (or will be) retired.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>All existing systems is (or will be) retired.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>A new generation of this type of systems is (or will be) developed from scratch.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Parts/components/modules of the future system are (or will be) reused from more than one of the existing systems.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Reused parts/components/modules required (or will require) only minor modifications</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>A significant amount of the existing systems are (or will be) reused in the future system</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>The functionality of the existing systems are equal.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>The quality of the existing systems are equal.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>At least some software parts/components/modules is (or will be) completely new</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Source code is (or will be) reused from one or more of the existing systems.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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…

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>…was based on technical considerations</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>…was based on considerations on time schedule</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>…was based on considerations for existing users</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>27</td>
<td>…was based on considerations concerning the parallel maintenance and evolution of existing systems</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>…was based on available staff and skills</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>…was based on politics</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>…was made by technicians</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>…was made by management</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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”.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>The existing systems will continue to be maintained, evolved and deployed completely separately.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>33</td>
<td>One of the existing systems is (or will be) evolved into a common system.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>34</td>
<td>One or more systems has been (or will be) retired.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>35</td>
<td>All existing systems is (or will be) retired.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>36</td>
<td>A new generation of this type of systems is (or will be) developed from scratch.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>Parts/components/modules of the future system are (or will be) reused from more than one of the existing systems.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>38</td>
<td>Reused parts/components/modules required (or will require) only minor modifications</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>39</td>
<td>A significant amount of the existing systems are (or will be) reused in the future system</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>40</td>
<td>The functionality of the existing systems are equal.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>The quality of the existing systems are equal.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>At least some software parts/components/modules is (or will be) completely new</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Source code is (or will be) reused from one or more of the existing systems.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comments</th>
<th></th>
</tr>
</thead>
</table>
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”.

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>None of the existing systems will be retired.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>One or more existing system will be retired.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>There will be a replacement system that covers all the lost functionality of retired system(s).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46</td>
</tr>
</tbody>
</table>

This decision was based on the opinions of...

<table>
<thead>
<tr>
<th>Opinion type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>customers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>users</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>developers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>marketing people</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
</tr>
</tbody>
</table>

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”.

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>None of the existing systems will be retired.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>One or more existing system will be retired.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>53</td>
</tr>
<tr>
<td>There will be a replacement system that covers all the lost functionality of retired system(s).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54</td>
</tr>
</tbody>
</table>

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”.

<table>
<thead>
<tr>
<th>Important aspect of backward compatibility</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>NA</th>
</tr>
</thead>
</table>
The future system needs to...

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>...support the way users currently work.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>...be backwards compatible with existing data.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>...be backwards compatible with existing surrounding tools.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>...be backwards compatible with installations of the existing systems.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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”.*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>The software of the existing systems have the same internal structure (architecture).</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>The parts/components/modules exchange data in the same ways in the existing systems.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>The existing systems interacts with the users in the same way.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>The existing systems have similar look-and-feel of the user interface.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>The existing systems contain software parts/components/modules with similar functionality.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>The hardware topology (networks, nodes) of the systems is similar.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>The design of the existing systems is based on the data model.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------------------------------------------------------------------------</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>The data models in the existing systems are similar.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>The implementations of data handling in the existing systems are similar.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>The existing systems are written in the same programming language.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>Communication between components/modules-parts in the existing systems is performed through certain interfaces.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>The existing systems use some technology to clearly encapsulate software components/modules-parts.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>The existing software use the same or similar technologies.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>The existing systems implement some domain standards.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>The existing systems implement the same domain standards.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>The existing systems were initially built in the same time period (e.g. decade).</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>The existing systems have evolved from the same system many years ago.</td>
<td>1 2 3 4 5 NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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”.

<table>
<thead>
<tr>
<th>Comments</th>
<th>Importance</th>
<th>Attention</th>
</tr>
</thead>
<tbody>
<tr>
<td>A small group of experts must be assigned early to evaluate the existing systems and describe alternative high-level strategies for the integration.</td>
<td>1 2 3 4 5 NA</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>Experience of the existing systems from many points of view must be collected.</td>
<td>1 2 3 4 5 NA</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>80</td>
<td>The future system should be described in terms of the existing systems.</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>82</td>
<td>The future system must contain more features than the existing systems</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>84</td>
<td>Decisions should wait until there is enough basis for making a decision</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>86</td>
<td>It is more important that decisions are made in a timely manner, even if there is not enough basis for making a decision</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>88</td>
<td>A strong project management is needed</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>90</td>
<td>All stakeholders must be committed to the integration</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>92</td>
<td>Management needs to show its commitment by allocating enough resources</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>94</td>
<td>The “grassroots” (i.e. the people who will actually do the hard and basic work) must be cooperative, both with management and each other</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>96</td>
<td>Formal agreements between sites must be made and honored (strictly obeyed)</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>98</td>
<td>A common development environment is needed</td>
<td>1 2 3 4 5 NA</td>
</tr>
<tr>
<td>100</td>
<td>There is a conflict between the integration efforts and other development efforts</td>
<td>1 2 3 4 5 NA</td>
</tr>
</tbody>
</table>
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.)

<table>
<thead>
<tr>
<th>Respondent ID</th>
<th>Question ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E1</td>
<td>E2</td>
<td>F2</td>
<td>F4</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>25</td>
<td>20</td>
<td>12</td>
<td>19</td>
<td>5</td>
<td>5</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>19</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Respondent ID</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Respondent ID</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>--------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>63</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>65</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>66</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>67</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>68</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>69</td>
<td>5</td>
<td>5</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>70</td>
<td>5</td>
<td>3</td>
<td>NA</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>71</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>72</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>73</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>74</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>NA</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>75</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>76</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>77</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>78</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>79</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>80</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>81</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>NA</td>
<td>1</td>
<td>2</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>82</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>83</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>4</td>
</tr>
<tr>
<td>84</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>85</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>86</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>NA</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>88</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>89</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>90</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>91</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>NA</td>
<td>2</td>
<td>NA</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>92</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>93</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>94</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>96</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>98</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>99</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Respondent ID</td>
<td>Question ID</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>NA</td>
<td>4</td>
<td>5</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>NA</td>
<td></td>
<td>NA</td>
<td>NA</td>
<td>1</td>
</tr>
</tbody>
</table>