Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 49 (2007) 419–444

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Software systems in-house integration: Architecture, process practices, and strategy selection

Rikard Land [*], Ivica Crnkovic

Department of Computer Science and Electronics, Mälardalen University, Box 883, SE-721 23 Vasteras, Sweden

Received 5 January 2006; received in revised form 15 June 2006; accepted 5 July 2006
Available online 22 August 2006

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.
© 2006 Elsevier B.V. All rights reserved.

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,

[*] Corresponding author. Tel.: +46 21 107035; fax: +46 21 103110.
E-mail addresses: rikard.land@mdh.se (R. Land), ivica.crnkovic@mdh.se (I. Crnkovic).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.07.002

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 20 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 3–4 months on each occasion, with 2 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 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

Table 1
Summary of the cases

	Organization	System domain	Goal	Information resources
A	Newly merged international company	Safety-critical systems with embedded software	New HMI[a] platform to be used for many products	*Interview:* project leader for "next generation" development project (I_A)
B	National corporation with many daughter companies	Administration of stock keeping	Rationalizing two systems within corporation with similar purpose	*Interview:* experienced manager and developer (I_B)
C	Newly merged international company	Safety-critical systems with embedded software	Rationalizing two core products into one	*Interviews:* leader for a small group evaluating integration alternatives (I_{Ca}); main architect of one of the systems (I_{Cb})
D	Newly merged international company	Off-line management of power distribution systems	Reusing HMI[a] for Data-Intensive Server	*Interviews:* architects/developers (I_{Da}, I_{Db}).
E1	Cooperation defense research institute and industry	Off-line physics simulation	Creating next generation simulation models from today's	*Interview:* project leader and main interface developer (I_{E1}) *Document:* protocol from startup meeting (D_{E1})
E2	Different parts of Swedish defense	Off-line physics simulation	Possible rationalization of three simulation systems with similar purpose	*Interview:* project leader and developer (I_{E2}) *Documents:* evaluation of existing simulation systems (D_{E2a}); other documentation (D_{E2b}, D_{E2c}, D_{E2d}, D_{E2e}, D_{E2f})
F1	Newly merged international company	Managing off-line physics simulations	Possible rationalization by using one single system	*Participation*: 2002 (R.L.) (P_{F1a}); currently (R.L.) (P_{F1b}). *Interviews:* architects/developers (I_{F1a}, I_{F1b}); QA responsible (I_{F1c}) *Documentation:* research papers (D_{F1a}); project documentation (D_{F1b})
F2	Newly merged international company	Off-line physics simulation	Improving the current state at two sites	*Interviews:* software engineers (I_{F2a}, I_{F2b}, I_{F2f}); project manager (I_{F2c}); physics experts (I_{F2d}, I_{F2e})
F3	Newly merged international company	Software issue reporting	Possible rationalization by using one single system	*Interview:* project leader and main implementer (I_{F3}) *Documentation:* miscellaneous related (D_{F3a}, D_{F3b})

[a] HMI, Human–Machine Interface.

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 Fig. 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

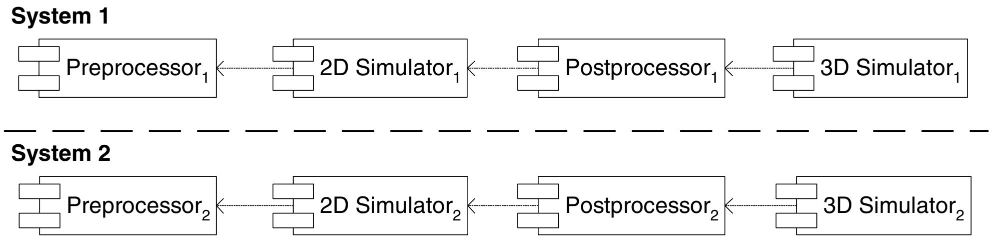


Fig. 1. The batch sequence architecture of the existing systems in case F2. Arrows denote dependency; data flows in the opposite direction.

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 (I_{F2a}:9, I_{F2c}:7,9, I_{F2d}:6,7, I_{F2e}:7).
- *3D simulator.* The plan for the (far) future is that the complete 3D simulator should be common, i.e. *Evolutionary Merge* (I_{F2a}:3, I_{F2c}:3, I_{F2f}:3). "X" physics is today handled by a new, commonly developed module that is used in both the Swedish and US 3D simulators (I_{F2e}:7). It has a new design, but there are similarities with the previous modules (I_{F2d}:7). In order to achieve this, new data structures and interfaces used internally have been defined and implemented from scratch (I_{F2e}:7, I_{F2f}:6,7); common error handling routines were also created from scratch (I_{F2e}: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 (I_{F2e}:7,8, I_{F2f}: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 (I_{F2e}: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 (I_{F1c}:6, I_{F1f}:6). Fig. 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 Fig. 3.

2.2.2. Other cases

This section presents the most relevant observations in each of the remaining cases.

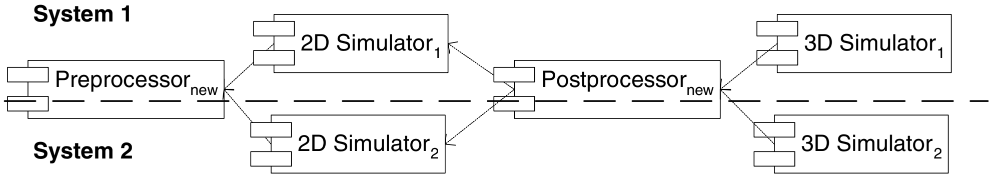


Fig. 2. The currently common and different parts of the systems in case F2.

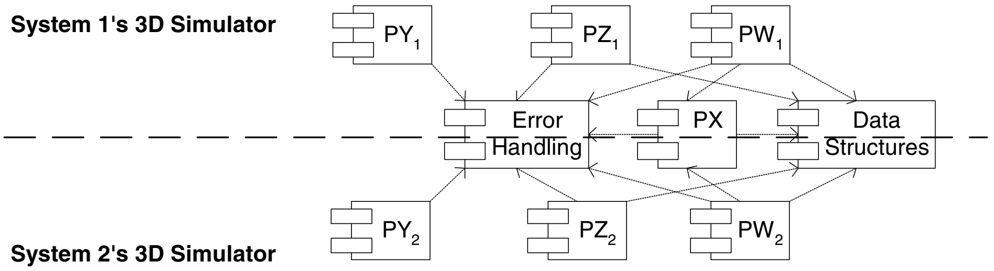


Fig. 3. The currently common and different parts (exemplified) of the 3D simulators in case F2.

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 (I_A: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 (I_A: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 (I_A: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 (I_A: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 (I_A:3,5). These two systems had very different underlying platforms: one was based on open source platforms and the other on commercial solutions (I_A: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 (I_A:3,5). Therefore, *Start from Scratch* was desired by the architects over *Choose One* and thus selected (I_A: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 (I_B: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 (I_B: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 (I_B:1,6,7,13). The difference in approach made *Merg*ing the systems infeasible, therefore the *Choose One* strategy was chosen (I_B:7). The total integrated system was stripped down piece by piece, and functionality rebuilt within the technological framework of the loosely integrated system (I_B:3,7). Many design ideas were however reused from the totally integrated system (I_B: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 (I_{Cb}:1,9). The systems' high-level structure were similar (I_{Ca}:7, I_{Cb}: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 (I_{Cb}:1,6,7). At the time of the company merger, new generations of these two systems were being developed and both were nearing release (I_{Ca}:1, I_{Cb}:1). Senior management first demanded reuse of one system's HMI and the other's underlying software within 6 months, i.e. *Rapid Merge*, which the architects considered unrealistic (I_{Ca}:6,8, I_{Cb}:6). The strict separation of software parts implied by being embedded in different hardware parts could make *Evolutionary Merge* possible, requiring 2 years. *Merge* was, according to the architects, the wrong way to go, and a better option would be to *Choose One* (I_{Ca}:6, I_{Cb}: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 (I_{Ca}:6). The delay caused an estimated loss of 1 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 (I_{Ca}:6, I_{Cb}: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 (I_{Ca}:12). Once this decision was made, reusing some components of the discontinued system into the other became easier (I_{Cb}: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" (I_{Cb}: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 (I_{Da}:1,3, I_{Db}: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 (I_{Da}: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* (I_{Da}:1, I_{Db}: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) (I_{Da}:1, I_{Db}: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, I_{Db}: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 (I_{Db}:6) and the high-level architecture, client–server (I_{Da}:7, I_{Db}:7,8); these similarities were partly due to a common ancestry some 20 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 5 years ago when the modern HMI was developed (I_{Db}: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, I_{Db}: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, D_{E2a}). 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 (D_{E2a}, 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 (I_{F1a}:1, I_{F1b}:1, I_{F1c}:1,2, D_{F1a}, P_{F1a}, P_{F1b}). An ambitious project was launched with the goal of evaluating the existing systems. There were differences in architectures, programming languages, technologies, and data models (I_{F1a}:6, I_{F1b}:6, I_{F1c}:6,7,9, D_{F1a}, P_{F1a}, P_{F1b}). It was not considered possible to discontinue development on the existing systems before a full replacement was available (I_{F1c}:6, D_{F1a}, P_{F1a}, P_{F1b}). 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}:3,6, 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 4 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 10-year-old version of a commercial system (I_{F3}:1). The two systems developed in-house where 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?" (I_{Cb}:7) is indeed a valid question, which will arise if a decision is not accompanied with priority and enough resources (I_{F1b}:3, I_{F1c}:6,9,11, P_{F1a}). Sometimes it might simply not be worth the effort to integrate – will the future savings

Table 2
The exclusion of possible strategies

		Start from Scratch	Choose One	Merge	
				Evolutionary	Rapid
Architectural Compatibility: "The similarity of existing systems is…"	"…very high"				
	"…modest"				■
	"…very small"			■	■
Retireability: "It is possible to retire…"	"…all"				
	"…some"	■			
	"…none"	■	■		

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.

through rationalization be larger than the integration efforts (I_{F1c}:9, I_{F2d}:3)? In case E2 there were very few resources available, which led to a very modest vision, in practice meaning no integration (I_{E2}: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 3.1 (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_{Ca}:6, I_{Cb}:6, I_{F1c}:6, P_{F1a}, P_{F1b}, 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_{Cb}: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_{Cb}: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_{Cb}: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 3.2 (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_{F2e}:6, I_{F2f}:6, I_F: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_A: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_{Ca}:6, D_{E2a}, I_{F3}:6). Case F2 involves complex scientific physics calculations, and the study of the existing systems' documentation of the implemented models was an important activity (I_{F2e}:6, I_{F2f}: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 (I_{F3}:11).

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

Proposition 3.3 (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 (I_{F2f}:6); in case F2, it was indeed considered a mistake to keep the old data format and adapt new development to it (I_{F2a}:9, I_{F2d}:7,9,11). The actual needs must be more important than to preserve the features of the existing systems (I_{F3}: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 (I_{F2f}: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 (I_{E1}:6); the organization was interested in Ada as such, which also contributed to this choice (I_{E1}: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 3.4 (Timely decisions). *Decisions must be made in a timely manner* (I_{Ca}: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 1 year of development money was wasted on parallel development, and the discontinued system has to be supported for years to come (I_{Ca}: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 3.5 (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 (I_{F2a}:9, I_{F2b}:9, I_{F2c}:3, I_{F2d}: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 (I_{F1b}: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 Section 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 Section 4.3 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_{E1}: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:

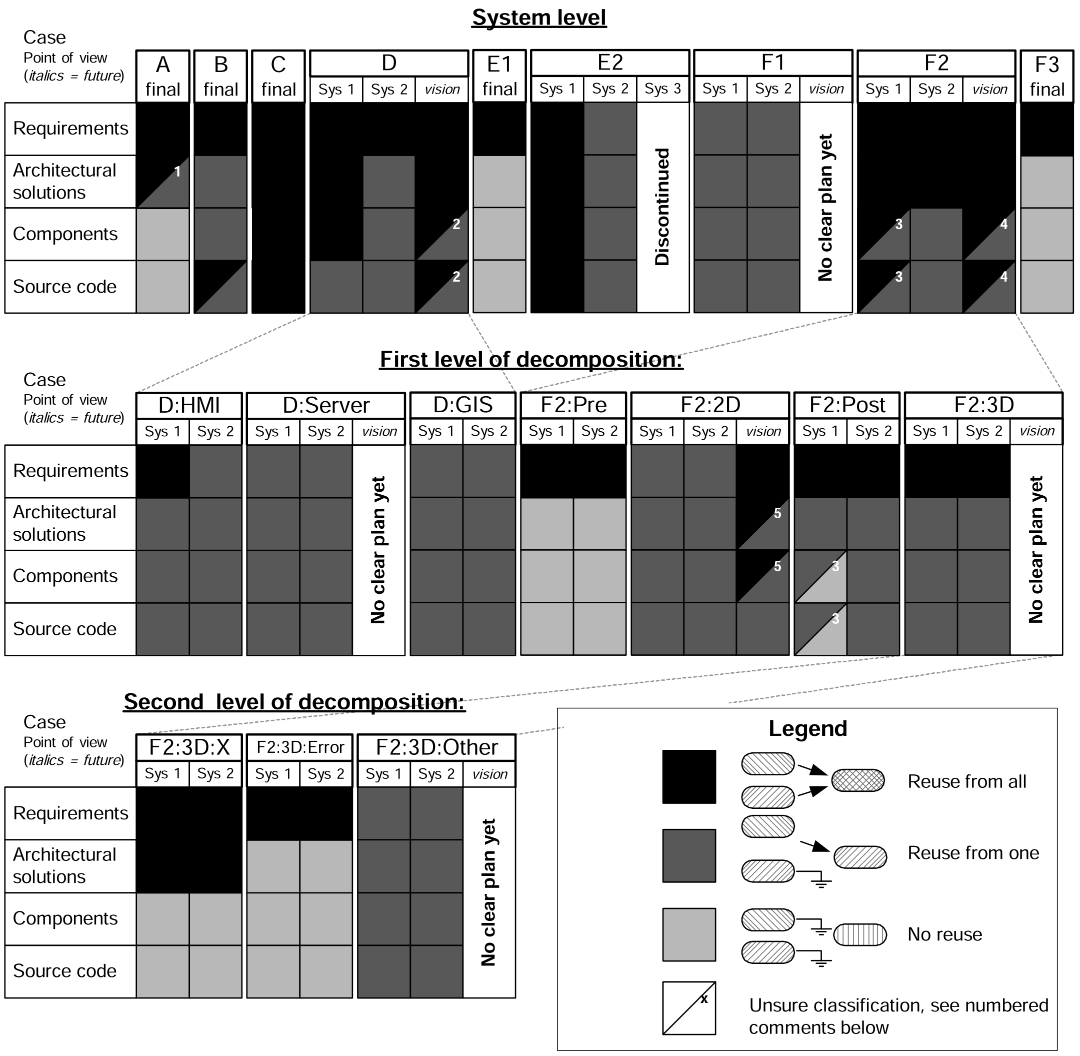
Starting from system level, if the architectural structures of the existing systems are similar and there are components with similar roles, then it is possible to hierarchically decompose these components and recursively consider observations 1 and 2. If, on the other hand, the structures are not similar 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 20 years or more (I_{Da}: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_{Cb}: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

Table 3
The types of reuse for the different artifacts in the cases



System level

Case / Point of view (*italics = future*)	A	B	C	D	D	D	E1	E2	E2	E2	F1	F1	F1	F2	F2	F2	F3
	final	final	final	Sys 1	Sys 2	*vision*	final	Sys 1	Sys 2	Sys 3	Sys 1	Sys 2	*vision*	Sys 1	Sys 2	*vision*	final
Requirements	■	■	■	■	■	■	■	■	▩	Discontinued	▩	▩	No clear plan yet	■	■	■	■
Architectural solutions	◪ 1	▩	■	■	▩	■	□	■	▩		▩	▩		■	■	■	□
Components	□	▩	■	■	▩	◪ 2	□	■	▩		▩	▩		◪ 3	▩	◪ 4	□
Source code	□	◪	■	▩	▩	◪ 2	□	■	▩		▩	▩		◪ 3	▩	◪ 4	□

First level of decomposition:

Case / Point of view (*italics = future*)	D:HMI	D:HMI	D:Server	D:Server	D:Server	D:GIS	D:GIS	F2:Pre	F2:Pre	F2:2D	F2:2D	F2:2D	F2:Post	F2:Post	F2:3D	F2:3D	F2:3D
	Sys 1	Sys 2	Sys 1	Sys 2	*vision*	Sys 1	Sys 2	Sys 1	Sys 2	Sys 1	Sys 2	*vision*	Sys 1	Sys 2	Sys 1	Sys 2	*vision*
Requirements	■	▩	▩	▩	No clear plan yet	▩	▩	■	■	▩	▩	■	■	■	■	■	No clear plan yet
Architectural solutions	▩	▩	▩	▩		▩	▩	□	□	▩	▩	◪ 5	▩	▩	▩	▩	
Components	▩	▩	▩	▩		▩	▩	□	□	▩	▩	◪ 5	◪ 3	▩	▩	▩	
Source code	▩	▩	▩	▩		▩	▩	□	□	▩	▩	▩	◪ 3	▩	▩	▩	

Second level of decomposition:

Case / Point of view (*italics = future*)	F2:3D:X	F2:3D:X	F2:3D:Error	F2:3D:Error	F2:3D:Other	F2:3D:Other	F2:3D:Other
	Sys 1	Sys 2	Sys 1	Sys 2	Sys 1	Sys 2	*vision*
Requirements	■	■	■	■	▩	▩	No clear plan yet
Architectural solutions	■	■	□	□	▩	▩	
Components	□	□	□	□	▩	▩	
Source code	□	□	□	□	▩	▩	

Legend

■	Reuse from all
▩	Reuse from one
□	No reuse
◪ x	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).

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, 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 (I_{Ca}:6, I_{Cb}: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" (I_{F3}: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 (I_{F1a}:6; D_{F1a}, P_{F1a}, P_{F1b}). In case F2, a new data model was defined and the existing systems adapted (I_{F2e}:7, I_{F2f}:6). In case F3, the three existing systems all implemented similar workflows, however the phases were different (I_{F3}: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" (I_{Cb}:6, also P_{F1a}). In case C, this caused an expensive delay as well as other problems (I_{Ca}:6,7), and in case F1 the architecture and outlined integration plan felt watered-down (D_{F1a}, I_{F1c}:6), and nothing happened to realize it (P_{F1a}, P_{F1b}).

5. 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 Section 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 (I_{Cb}:1,7, I_{Cb}:6, I_{E1}:4, I_{F2e}:6, I_{F2a}: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 (I_{Cb}:7, I_{Cb}:6); however development did continue until both systems were released, which led to lots of extra costs and problems (I_{Cb}: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" (I_{Db}:7). This was influential when deciding to keep this HMI (I_{Db}:7,8). In case B, the loose integration of one of the systems which would ensure long-term evolvability, was influential when choosing system (I_B: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 (I_{Db}: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 (I_{F2e}:6, I_{F2a}: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) (I_{Ca}: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 to 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 (I_{Ca}: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 (D_{F1}, [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 inhouse 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 6.1 (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_{Ca}:11, I_{Da}:3, I_{Db}: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_{Ca}:6), and in case D assignment of development money to delivery projects is apparently not in line with the long-term integration goals (I_{Da}:3, I_{Db}:6). “Projects including costly project specific development are not punished. The laws of market economy are disabled”. (I_{Da}: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 6.2 (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_{F2d}: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 6.3 (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_{Db}: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_{Db}:6,11). The overall goals must be made clear to the grassroots to gain the necessary commitment and “buy-in” (I_{Cb}:11, I_{F1b}:11). The “not invented here syndrome” is dangerous for cooperation (I_{Db}: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 6.4 (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 6.5 (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 6.6 (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_{F2f}: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 6.7 (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_{Da}:12, I_{Db}: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_{F2e}: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_{F2f}: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_{F2b}:6, I_{F2c}:9, I_{F2d}: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_{Ca}: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_{F3}: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 an *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.

Table 4
Summary of the possible and desired strategies in the cases

Strategy / Case	Start from Scratch	Choose One	Merge	
			Evolutionary	Rapid
A	✓		■	■
B	■	✓	■	■
C (initial)	■	■		■ ✓
C (final)	■	✓		■
***D**	■ (?)	■ (?)	✓	■ (?)
D_{HMI}	■	✓	■	■
***D_{Server}**	■ (?)	■ (?)	✓	■ (?)
E1	✓			■
E2	■	✓		■
F1 (initial)	■	■	■ ✓	■
***F1 (second decision)**	■	■ ✓	■	■
***F2**	■	■	✓	■
$F2_{Pre}$	✓		■ (?)	■
***$F2_{2D}$**		✓		■
$F2_{Post}$	✓		■	■
***$F2_{3D}$**	■	■	✓	■
F3	✓		■ (?)	■

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

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,Zetc. 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 α_X from system X, component α_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 he 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.

Acknowledgments

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.

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. For example, 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? For example, 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?

References

[1] G.D. Abowd, R. Allen, D. Garlan, Using style to understand descriptions of software architecture, in: Proceedings of The First ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1993.
[2] P. Abrahamsson, The Role of Commitment in Software Process Improvement, Ph.D. Thesis, Department of Information Processing Science, University of Oulu, 2005.
[3] R. Allen, A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, 1997.
[4] A. April, J. Huffman Hayes, A. Abran, R. Dumke, Software Maintenance Maturity Model (SMmm): the software maintenance process model, Journal of Software Maintenance and Evolution: Research and Practice 17 (3) (2005) 197–223.
[5] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003, ISBN 0-321-15495-9.
[6] K. Beck, EXtreme Programming EXplained: Embrace Change, Addison Wesley, 1999, ISBN 0201616416.
[7] V. Berzins, Software merge: semantics of combining changes to programs, ACM Transactions on Programming Languages and Systems (TOPLAS) 16 (6) (1994) 1875–1903.
[8] B. Boehm, Spiral Development: Experience, Principles and Refinements, CMU/SEI-2000-SR-008, Software Engineering Institute, Carnegie Mellon University, 2000.
[9] B. Boehm, R. Turner, Using risk to balance agile and plan-driven methods, IEEE Computer 36 (6) (2003) 57–66.
[10] M.L. Brodie, M. Stonebraker, Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach, Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, 1995. ISBN 1558603301.
[11] F.P. Brooks, No silver bullet, in: The Mythical Man-Month – Essays On Software Engineering, 20th Anniversary ed., Addison-Wesley Longman, 1995, ISBN 0201835959.
[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons, 1996, ISBN 0-471-95869-7.
[13] E. Carmel, Global Software Teams – Collaborating Across Borders and Time Zones, Prentice-Hall, 1999, ISBN 0-13-924218-X.
[14] E. Carmel, R. Agarwal, Tactical approaches for alleviating distance in global software development, IEEE Software 18 (2) (2001) 22–29.
[15] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Evaluating Software Architectures, Addison-Wesley, 2001, ISBN 0-201-70482-X.
[16] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002, ISBN 0-201-70372-6.
[17] CMMI Product Team, Capability Maturity Model® Integration (CMMI SM), Version 1.1, CMU/SEI-2002-TR-011, Software Engineering Institute (SEI), 2002.
[18] H. Courtney, 20|20 Foresight: Crafting Strategy in an Uncertain World, Harvard Business School Press, 2001, ISBN 1-57851-266-2.
[19] I. Crnkovic, Component-based software engineering – new challenges in software development, in: Software Focus, John Wiley & Sons, 2001.

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

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

[22] L. Davis, D. Flagg, R.F. Gamble, C. Karatas, Classifying interoperability conflicts, in: Proceedings of Second International Conference on COTS-Based Software Systems, LNCS 2580, Springer-Verlag, 2003, pp. 62–71.

[23] L. Davis, R. Gamble, J. Payton, G. Jónsdóttir, D. Underwood, A Notation for Problematic Architecture Interactions, ACM SIGSOFT Software Engineering Notes 26 (5) (2001).

[24] T. DeMarco, T. Lister, Peopleware: Productive Projects and Teams, second ed., Dorset House Publishing, 1999, ISBN 0-932633-43-9.

[25] E.H. Ferneley, Design metrics as an aid to software maintenance: an empirical study, Journal of Software Maintenance: Research and Practice 11 (1) (1999) 55–72.

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

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

[28] D. Garlan, R. Allen, J. Ockerbloom, Architectural Mismatch: Why Reuse is so Hard, IEEE Software 12 (6) (1995) 17–26.

[29] D. Garlan, M. Shaw, An introduction to software architecture, Advances in Software Engineering and Knowledge Engineering vol. I (1993).

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

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

[32] J.D. Herbsleb, D. Moitra, Global software development, IEEE Software 18 (2) (2001) 16–20.

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

[34] G. Hofstede, Cultures and Organizations: Software of the Mind, second ed., McGraw-Hill, 2004, ISBN 0071439595.

[35] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Pearson Education, 2004, ISBN 0321200683.

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

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

[38] L. Jakobsson, B. Christiansson, I. Crnkovic, Component-based development process, in: I. Crnkovic, M. Larsson (Eds.), Building Reliable Component-Based Software Systems, Artech House, 2002, ISBN 1-58053-327-2.

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

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

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

[42] R. Kazman, M. Barbacci, M. Klein, J. Carriere, Experience with performing architecture tradeoff analysis method, in: Proceedings of The International Conference on Software Engineering, New York, 1999, pp. 54–63.

[43] S. Komi-Sirviö, M. Tihinen, Lessons learned by participants of distributed software development, Knowledge and Process Management 12 (2) (2005) 108–122.

[44] M. Korhonen, T. Mikkonen, Assessing systems adaptability to a product family, in: Proceedings of International Conference on Software Engineering Research and Practice, CSREA Press, 2003.

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

[46] P. Kruchten, The Rational Unified Process: An Introduction, second ed., Addison-Wesley, 2000, ISBN 0-201-70710-1.

[47] C.W. Krueger, Software reuse, ACM Computing Surveys 24 (2) (1992) 131–183.

[48] R. Land, L. Blankers, S. Larsson, I. Crnkovic, Software systems in-house integration strategies: merge or retire – experiences from industry, in: Proceedings of Software Engineering Research and Practice in Sweden (SERPS), 2005.

[49] R. Land, J. Carlson, I. Crnkovic, S. Larsson, A method for exploring software systems merge alternatives, in: Proceedings of submitted to Quality of Software Architectures (QoSA) (will otherwise be published as a technical report, the paper can be found at <www.idt.mdh.se/~rld/temp/MergeMethod.pdf>), 2006.

[50] R. Land, I. Crnkovic, Software systems integration and architectural analysis – a case study, in: Proceedings of International Conference on Software Maintenance (ICSM), IEEE, 2003.

[51] R. Land, I. Crnkovic, Existing approaches to software integration – and a challenge for the future, in: Proceedings of Software Engineering Research and Practice in Sweden (SERPS), Linköping University, 2004.

[52] R. Land, I. Crnkovic, S. Larsson, Concretizing the vision of a future integrated system – experiences from industry, in: Proceedings of 27th International Conference Information Technology Interfaces (ITI), IEEE, 2005.

[53] R. Land, I. Crnkovic, S. Larsson, L. Blankers, Architectural concerns when selecting an in-house integration strategy – experiences from industry, in: Proceedings of 5th IEEE/IFIP Working Conference on Software Architecture (WICSA), IEEE, 2005.

[54] R. Land, I. Crnkovic, S. Larsson, L. Blankers, Architectural reuse in software systems in-house integration and merge – experiences from industry, in: Proceedings of First International Conference on the Quality of Software Architectures (QoSA), Springer, 2005.

[55] R. Land, I. Crnkovic, C. Wallin, Integration of software systems – process challenges, in: Proceedings of Euromicro Conference, 2003.

[56] R. Land, M. Lakotic, A tool for exploring software systems merge alternatives, in: Proceedings of International ERCIM Workshop on Software Evolution, 2006.

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

[58] R. Land, S. Larsson, I. Crnkovic, Processes patterns for software systems in-house integration and merge – experiences from industry, in: Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI), 2005.

[59] R. Land, P. Thilenius, S. Larsson, I. Crnkovic, 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.

[60] R. Land, P. Thilenius, S. Larsson, I. Crnkovic, Software in-house integration – quantified experiences from industryProceedings of 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI), IEEE, 2006.

[61] S. Larsson, I. Crnkovic, Case study: software product integration practices, in: Proceedings of 6th International Conference on Product Focused Software Process Improvement (PROFES), Springer, 2005.

[62] M.M. Lehman, J.F. Ramil, Rules and tools for software evolution planning and management, Annals of Software Engineering 11 (1) (2001) 15–44.

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

[64] A. Maxwell Joseph, Understanding and validity in qualitative research, Harvard Educational Review 62 (3) (1992) 279–300.
[65] S. McConnell, Rapid Development, Taming Wild Software Schedules, Microsoft Press, 1996, ISBN 1-55615-900-5.
[66] T. Mens, A state-of-the-art survey on software merging, IEEE Transactions on Software Engineering 28 (5) (2002) 449–462.
[67] D.G. Messerschmitt, C. Szyperski, Software Ecosystem: Understanding an Indispensable Technology and Industry, MIT Press, 2003, ISBN 0-262-13432-2.
[68] C. Meyers, P. Oberndorf, Managing Software Acquisition: Open Systems and COTS Products, Addison-Wesley, 2001, ISBN 0201704544.
[69] C. Meyers, T. Oberndorf, Open Systems: The Promises and the Pitfalls, Addison-Wesley, 1997, ISBN 0-201-70454-4.
[70] P. Oman, J. Hagemeister, D. Ash, A Definition and Taxonomy for Software Maintainability, SETL Report 91-08-TR, University of Idaho, 1991.
[71] B. Omelayenko, Integration of product ontologies for B2B marketplaces: a preview, ACM SIGecom Exchanges 2 (1) (2000) 19–25.
[72] D.L. Parnas, Software aging, in: Proceedings of the 16th International Conference on Software Engineering, IEEE Press, 1994, pp. 279–287.
[73] D.E. Perry, Laws and principles of evolution, in: Proceedings of International Conference on Software Maintenance (ICSM), IEEE, 2002, p. 70.
[74] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.
[75] M.E. Porter, Competitive Strategy: Techniques for Analyzing Industries and Competitors, Free Press, 1998, ISBN 0684841487.
[76] J.S. Poulin, Measuring Software Reuse: Principles, Practices, and Economic Models, Addison-Wesley, 1997, ISBN 0-201-63413-9.
[77] C. Robson, Real World Research, second ed., Blackwell Publishers, 2002, ISBN 0-631-21305-8.
[78] N. Rozanski, E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Addison-Wesley, 2005, ISBN 0-321-11229-6.
[79] W.A. Ruh, F.X. Maginnis, W.J. Brown, Enterprise Application Integration, A Wiley Tech Brief, John Wiley & Sons, 2000, ISBN 0471376418.
[80] K. Rönkkö, Making Methods Work in Software Engineering: Method Deployment – as a Social Achievement, Ph.D. Thesis, Blekinge Institute of Technology, 2005.
[81] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects, Wiley Series in Software Design Patterns, ISBN 0-471-60695-2, John Wiley & Sons Ltd., 2000.
[82] B. Seaman, Qualitative methods in empirical studies of software engineering, IEEE Transactions on Software Engineering 25 (4) (1999) 557–572.
[83] M.T. Sewell, L.M. Sewell, The Software Architect's Profession – An Introduction, Software Architecture Series, Prentice Hall PTR, 2002. ISBN 0-13-060796-7.
[84] J. Stapleton, DSDM – Dynamic Systems Development Method, Pearson Education, 1997, ISBN 0-201-17889-3.
[85] M. Stonebraker, J.M. Hellerstein, Content integration for E-business, ACM SIGMOD Record 30 (2) (2001) 552–560.
[86] A. Strauss, J.M. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, second ed., Sage Publications, 1998, ISBN 0803959400.
[87] C. Szyperski, Component Software – Beyond Object-Oriented Programming, second ed., Addison-Wesley, 2002, ISBN 0-201-74572-0.
[88] A.A. Thompson Jr., A.J. Strickland III, Strategic Management: Concepts and Cases, 11th ed., Irwin/McGraw-Hill, 1999, ISBN 0-07-303714-1.
[89] C. van der Westhuizen, A. van der Hoek, Understanding and propagating architectural change, in: Proceedings of Third Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3), Kluwer Academic Publishers, 2002, pp. 95–109.
[90] J. van Gurp, J. Bosch, Design erosion: problems and causes, Journal of Systems & Software 61 (2) (2002) 105–119.
[91] K.C. Wallnau, S.A. Hissam, R.C. Seacord, Building Systems from Commercial Components, Addison-Wesley, 2001, ISBN 0-201-70064-6.
[92] P. Wegner, Interoperability, ACM Computing Surveys 28 (1) (1996).
[93] J.C. Wileden, A. Kaplan, Software interoperability: principles and practice, in: Proceedings of 21st International Conference on Software Engineering, ACM, 1999, pp. 675–676.
[94] WWISA, Worldwide Institute of Software Architects, URL: <http://www.wwisa.org>, 2002.
[95] R.K. Yin, Case Study Research: Design and Methods, third ed., Sage Publications, 2003, ISBN 0-7619-2553-8.