Architectural Reuse in Software Systems In-house Integration and Merge – Experiences from Industry

Rikard Land[1], Ivica Crnković[1], Stig Larsson[1], and Laurens Blankers[1,2]

[1] Mälardalen University,
Department of Computer Science and Electronics,
PO Box 883, SE-721 23 Västerås, Sweden
[2] Eindhoven University of Technology,
Department of Mathematics and Computing Science,
PO Box 513, 5600 MB Eindhoven, Netherlands
{rikard.land, ivica.crnkovic, stig.larsson,
laurens.blankers}@mdh.se
http://www.idt.mdh.se/{~rld, ~icc}

Abstract. When organizations cooperate closely, for example after a company merger, there is typically a need to integrate their in-house developed software into one coherent system, preferably by reusing from all of the existing systems. The parts that can be reused may be arbitrarily small or large, ranging from code snippets to large self-containing components. Not only implementations can be reused however; sometimes it may be more appropriate to only reuse experiences in the form of architectural solutions and requirements. In order to investigate the circumstances under which different types of reuse are appropriate, we have performed a multiple case study, consisting of nine cases. Our conclusions are, summarized: reuse of components from one system requires reuse of architectural solutions from the same system; merge of architectural solutions cannot occur unless the solutions already are similar, or if some solutions from one are incorporated into the other. In addition, by hierarchically decomposing the systems we make the same observations. Finally, among the cases we find more architectural similarities than might had been expected, due to common domain standards and common solutions within a domain. Although these observations, when presented, should not be surprising, our experiences from the cases show that in practice organizations have failed to recognize when the necessary prerequisites for reuse have not been present.

1 Introduction

Given two high-quality systems with similar purpose and features, how can you create one single system? Can they somehow be merged? In other words, is it possible to reuse the best out of the existing systems and reassemble a new, future system of perhaps even higher quality? This is the challenge and desire of many organizations in today's era of company mergers and other types of close collaborations; this may even be the reason for acquiring a competitor in the first place. The software may be the core products of the companies, or some support systems for the core business. If

R. Reussner et al. (Eds.): QoSA-SOQUA 2005, LNCS 3712, pp. 123 – 139, 2005.

the software systems are mainly used in-house, performing further evolution and maintenance of two systems in parallel seems wasteful. If the software systems are products of the company, it makes little sense to offer customers two similar products. In either case, the organization would ideally want to take the best out of the existing systems and integrate or merge them with as little effort as possible, i.e. reusing whatever can be reused when building a future system.

However, it can be expected that "architectural mismatch" [8] makes this task difficult. Combining parts of two systems built under different assumptions, following different philosophies, would likely violate the conceptual integrity [3] of the system. It has been observed that qualities largely are determined by the architecture of a system [2], but no matter what the quality of the existing systems are, integrating or merging them – if possible at all – does not necessarily lead to a new high-quality system. There seems to be a range of possibilities in practice, and the choices are influenced by many factors. Possibilities include a tight merge where the old systems are no longer distinguishable, a looser integration, to discontinue some systems and evolve others, or even to not integrate at all but rather start a new development effort, or even (just to be exhaustive about the alternatives) doing nothing but let the existing systems live side by side. The current paper outlines the prerequisites for a tighter merge, in terms of the existing systems (other important points of view are e.g. processes, people, organization, and culture). That is: under what circumstances it is possible and feasible to reuse parts of the existing systems, and when is it more appropriate to only reuse experiences and implement something new?

In order to investigate this we have carried out a multiple case study [21]. It contains nine cases from six organizations working in different domains. The cases are situations where two or more systems were found to overlap and the intention was to create a new system for the future. This paper takes the viewpoint of reuse, and two specific questions are, within the context outlined:

Q1. Which are common experiences (good and bad) concerning reuse when merging two or more systems?
Q2. To what extent are the lessons learned from these experiences possible to generalize into recommendations for other organizations?

The rest of the paper describes the cases and provides answers to these questions. Section 1.1 describes related work, section 1.2 describes the methodology used in the research, and section 1.3 introduces the cases. Section 2 categorizes reuse and section 3 presents reuse in the cases. Section 4 consolidates the observations from all cases, summarized according to the categorization given. Section 5 concludes the paper by summarizing the observations made and outlining future work.

1.1 Related Work

Software integration may mean many different things, and there is much literature to be found. Three major fields of software integration are component-based software [20], open systems [16], and Enterprise Application Integration, EAI [17]. In a previous survey of existing approaches to software integration [13], we found no existing literature that directly addresses the context of the present research: integration or merge of software *controlled and owned within an organization*.

Software reuse usually means finding existing software pieces possible to use in a new situation [10,12]. In the context of the present paper, the procedure is rather the opposite: given two (or more) systems to reuse from, what can be reused? In addition, reuse traditionally concerns reusing implementations, while the cases also mention reuse of experiences, i.e. of requirements and known architectural and design solutions.

1.2 Research Methodology

The multiple case study [21] consists of nine cases from six organizations that have gone through an integration process. Our main data source has been interviews, but in some cases we also had access to certain documentation. In one case (F1) one of the authors (R.L.) also participated as an active member. Details regarding the research design and material from the interviews are available in a technical report [14].

1.3 Overview of the Cases

The cases come from different types and sizes of organizations operating in different domains, the size of the systems range from a maintenance and development staff of a few people to several hundred people, and the types of 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 labelled A, B, etc. Cases E1, E2, F1, F2, and F3 occurred within the same organizations (E and F). For the 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. The complete copied out interview notes, and details about the participation activities and documents used are found in [14]. In the present paper, we have provided explicit pointers into this source of data.

2 Categorizing Reuse

This section first describes development artefacts (i.e. not only implementations) that can be reused that were mentioned in the cases (section 2.1) followed by a presentation of three basic reuse types that can apply to each of these artefacts (section 2.2).

2.1 What Software Artefacts Can Be Reused?

Although software reuse traditionally means reuse of implementations [12], the cases repeatedly indicate reuse of experience even if a new generation is implemented. In order to capture this, we have chosen to enumerate four types of artefacts that can be

reused: requirements, architectural solutions (structure and framework), components and source code. The first two means reuse of concepts and experiences, and the two latter reuse of implementations. We have chosen the terms "component" and "framework", although aware that the terms is often given more limited definitions than is intended here, "component" in e.g. the field of Component-Based Software Engineering [19], and "framework" in e.g. object oriented frameworks [6,9] and component based frameworks [5].

- **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 needs improvement compared to the current state of the existing systems. (Not discussed in the present paper is the important 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:
  - *Structure* (the roles of components and relations between them), in line with the definition given e.g. by Bass et al [2]. Reusing structure would to a large part explicitly recognize architectural and design patterns and styles [1,4,7,18].
  - *Framework.* A definition suitable for our purposes is an "environment which defines components, containing certain rules to which the components must adhere to be considered components (it can be compliance to component models, or to somewhat vaguer definitions)". A framework embodies these rules in the form of an implementation enforcing and supporting some important 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.
- **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 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 "merged". We can thus talk about a hierarchical decomposition of systems. Large systems could potentially have several hierarchical levels.

Table 1. Summary of the cases

	Organization	**System Domain**	**Goal**	**Information Resources**
A	Merged international company	Safety-critical systems with embedded software	New Human-Machine Interface (HMI) platform to be used for many products	*Interview:* project leader for "next generation" development project (I_A)
B	Organization within large international enterprise	Administration of stock keeping	Rationalizing two systems within corporation with similar purpose	*Interview:* experienced manager and developer (I_B)
C	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	Merged international company	Off-line management of power distribution systems	Reusing Human-Machine Interface 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	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})
F2	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	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})

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, so an exact match is not necessary (and would be highly unlikely). For example, if two systems or components write similar info to a file, differences in syntax can be overcome with reasonable effort, and the interfaces can be considered compatible. However, reuse of interfaces also requires semantic compatibility, 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 artefacts is discussed in the present paper, the focus is on reuse of architectural solutions and components, and on the recursive (hierarchical) decomposition process.

2.2 Possible Primitive Types of Reuse in Software Merge

Assuming there are two systems to be integrated, there are three basic possibilities of reuse: *a)* reuse from both existing systems, *b)* reuse from only one of the existing systems, and *c)* reuse nothing. See Fig. 1. In reality there may be more than two existing systems (I_A:1, I_{E1}:1, I_{E2}:1, D_{F2a}, I_{F3}:1), and more reuse alternatives can easily be constructed by combining these primitive reuse types, i.e. reuse from one, or two, or …, or *n*-1 of the *n* existing systems. For simplicity, we only discuss this in text in connection to the cases.

Different types of reuse can be applied at each of the above mentioned/enumerated artefacts. For example, requirements might be reused from all systems (type *a*), but only the architecture and components of one is evolved (type *b*). An important pattern to search for in the cases is how different types of reuse for different artefacts 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?

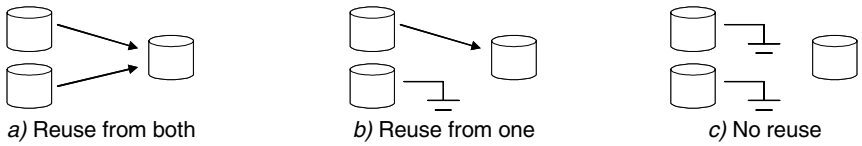


Fig. 1. The three basic types of reuse in the integration context

We want to emphasize that this is a very simple way of describing a complex phenomenon that cannot capture everything. For example, the focus of the paper is on reuse, not new influences. A problem with the three simple types 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. However, it would be problematic to describe different amounts of reuse – what would "reuse of 34% of the architectural solutions of system A" mean? This difficulty is taken into account in the analysis (section 4), by discussing the implications of classifying each case into either reuse type.

3 Reuse in the Cases

This section will present the type of reuse applied to the different artefacts in all of the nine cases, including considerations and motivations. One of the cases (case F2) is described in depth, followed by more summarized descriptions of the others. More details for all cases can be found in a technical report [14].

The motivation for selecting case F2 for the in-depth description is that being halfway into full integration it represents all types of reuse, at different hierarchical levels. 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 taken part of information not formalized through interview notes.

3.1 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 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. 2.

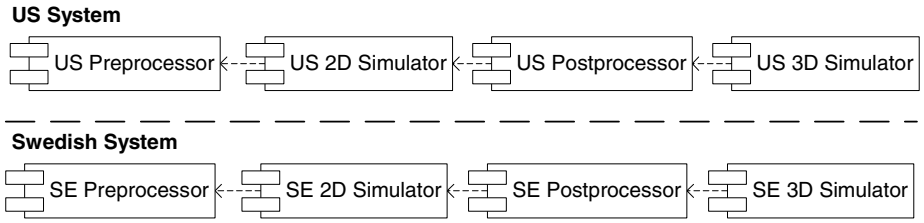


Fig. 2. The batch sequence architecture of the existing systems in case F2. Arrows denote dependency; data flows in the opposite direction.

The 3D simulator contains several modules modelling different aspects of the physics involved. One of these modules, which we can call "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 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 the US, a more sophisticated methodology for their 2D simulations was desired, a methodology already implemented in the Swedish

system (I_{F2a}:3). In the Swedish system on the other hand, fundamental problems with their model had also been experienced (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 US customers, the other common among Swedish 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 the "X" module in the 3D simulator 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_{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.** By evolving the US simulator, a new 2D simulator is being developed which will replace the existing 2D simulators (I_{F2a}:9, I_{F2b}:7, I_{F2d}:7,8). It will reuse a calculation methodology from the Swedish system (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/assumed that the old Swedish post-processor, with three layers written in different languages, would be the starting point, based on engineering judgments (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, 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_{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 framework 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).

Fig. 3 shows the current states of the systems. Although there are two 3D simulators, some internal parts are common; this illustrates the hierarchical decomposition described in section 2.1 and is visualized in Fig. 4.

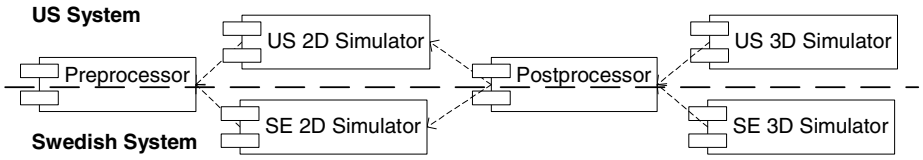


Fig. 3. The currently common and different parts of the systems in case F2

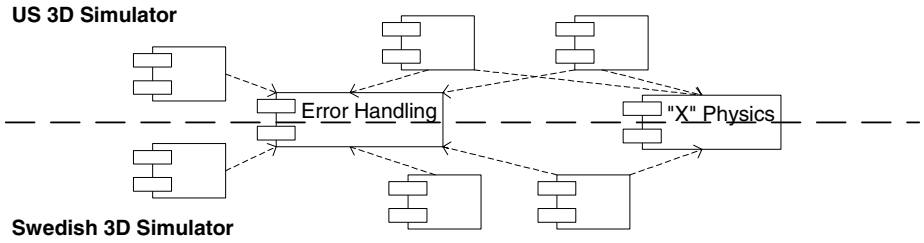


Fig. 4. The currently common and different parts (only exemplified) of the 3D simulators in case F2

3.2 Other Cases

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

Case A. Each of the previous separate companies had developed software human-machine interfaces (HMIs) for their large hardware products (I_A:1,2). To rationalize, it was decided that a single HMI should be used throughout the company (I_A: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 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).

Case B. One loosely integrated information system had been customized and installed at a number of daughter companies within a large enterprise (I_B:1). In one such 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 total integrated system was stripped down piece by piece, and functionality rebuilt within the 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. Two previously competing safety-critical and business-critical products with embedded software had to be integrated (I_{Ca}:1, I_{Cb}:1). 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 mechanisms such as failover, supporting different (natural) languages, and error handling, as well as a fundamental difference between providing an object model or being functionally oriented (I_{Cb}:1,6,7). Higher management first demanded reuse of one system's HMI and the others underlying software in a very short time, which by the architect's was considered unrealistic and the wrong way to go (I_{Ca}:6,8, I_{Cb}:6). The final decision was to retire one of the systems and continue development of the other (I_{Ca}:6, I_{Cb}:6). 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 was considered "modest size" compared to rewrite, although "the solutions were not the technically most elegant" (I_{Cb}:7).

Case D. After the company merger between a US and a European company 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). As the US system's HMI was considered out of fashion, two major improvements were made: the European HMI was reused, and a commercial GIS tool was acquired (instead of the European data engineering tool) (I_{Da}:1, I_{Db}:3,8). Reuse of the European HMI was possible thanks to its component-based architecture (I_{Da}:1, I_{Db}:3,7,8) and 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). The similarities were partly due to a common ancestry some twenty years earlier (I_{Da}:1). In the European HMI, a proprietary component framework had been built and it has been possible to transfer some functionality from the US HMI by adding and modifying components (I_{Db}:7). The servers are still different and the company markets two different systems (I_{Da}:12, I_{Db}:7).

Case E1. A number of existing simulation models were 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). Many ideas were reused, and transforming some existing SIMULA code to Ada was quite easy (I_{E1}:7).

Case E2. A certain functional overlap among three simulation systems was identified (I_{E2}:1, D_{E2a}). Due to very limited resources, one of these was retired, and the only integration between the remaining two has been reuse of the graphical user interface of one into the other (I_{E2}:6). Even though the systems belong to the same narrow domain, the same language were used and the integration was very loose, this reuse into the other system required more effort than expected, due to differences in input data formats, log files, and the internal model (I_{E2}:7).

Case F1. After a company merger there was a wish to integrate the software simulation environment within the company, focused around the major 3D simulators used, but also including a range of user interfaces 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 three existing systems; the final decision was to make the system share data in a common database (I_{F1a}:3, I_{F1c}:3, D_{F1a}, P_{F1a}). However, nothing has yet happened to implement it; it appears as this solution was perceived as a compromise by all involved (I_{F1c}:6, P_{F1a}, P_{F1b}). Subsequent meetings has not led to any tangible progress, and four years after the company merger there are still discussions on how to arrive at a future integrated environment, although even this goal itself is being questioned by some of the people concerned (I_{F1b}:3,9, I_{F1c}:6,9). A difficulty has been to integrate the existing data models (I_{F1a}:6, I_{F1b}:6, I_{F1c}:6,7,9, D_{F1a}, P_{F1a}, P_{F1b}). Part of the problem might also be that the scope of such a future system is unclear; discussions include the whole software environment at the departments (I_{F1b}:6,9, I_{F1c}:1, P_{F1b}).

Case F3. Three different software systems for tracking software issues (errors, requests for new functionality etc.) were used at three different sites within the company, two developed in-house and one being a ten-year old version of a commercial system (I_{F3}:1). 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 should be acquired and customized to support these (I_{F3}:6).

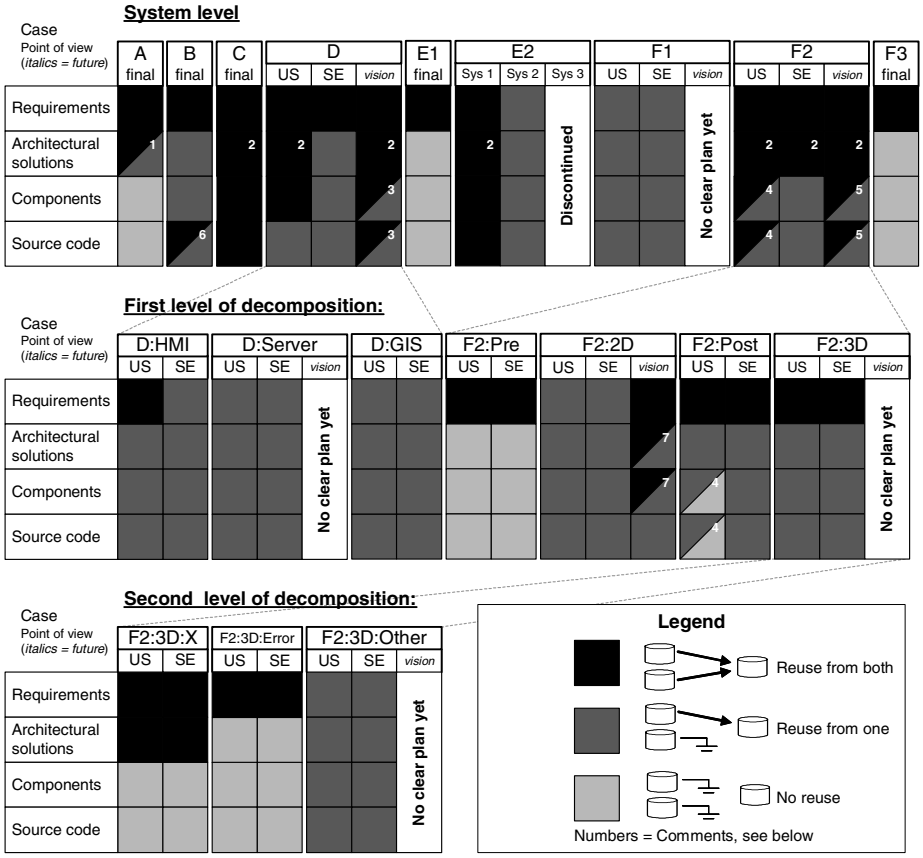
4 Analysis

The cases are summarized in Table 2. In four cases (D, E2, F1, and F2) the systems are not (yet) integrated or merged completely, and there are still two (or more) separate systems deployed, sharing some common parts. This means that if system X has reused something from system Y but not the other way around, and the two systems are still deployed separately, we would have instances of reuse type *a*, "reuse from all" (from X's point of view) and type *b*, "reuse from one" (from Y's point of view). In order to be able to describe this in terms of the "primitive" reuse types described above, reuse is considered from the point of view of the currently deployed systems (where applicable) as well as the future envisioned system (where applicable). 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 consider us to have enough material to do so (for case F2 in two levels); for most of the others, this did not make sense when reuse from more than one did not occur.

1. Architectural solutions were reused mainly from one, with heavy influence from one of (several) other systems.
2. The systems had already similar architecture, and the integrated system have thus reused the same solutions from two systems.
3. It is unknown what will be reused in the future integrated system.

4. One component (the post-processor) started out as an attempt to reuse from the Swedish system, but in the end only a small fraction of the original component was left and should probably be considered source code reuse.
5. It is unknown what will be reused in the future integrated system. Comment 5 also applies.
6. It is unsure whether any source code was reused from the retired system (not enough information).
7. The future systems will be an evolution of the US system, while incorporating a methodology from the Swedish system; it is not known whether this means reuse of certain architectural solutions and components (the latter seems unlikely).

Table 2. The types of reuse for the different artefacts in the cases



In the table, for each system we have listed the four artefacts 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 and some other fields have been marked with a number indicating a text comment (to be found below the table).

Based on Table 2, 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 8). 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.

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 (comment 2). 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 [8]. 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 high quality of both systems, and the very 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). We believe that by considering the roles of the components, it might be possible to find similarities and possibilities for component reuse, even if the structure of the components is dissimilar; this however remains a topic for further research.

Observation 4. In several of the cases, the architectures were similar (cases C, D, F2, and possibly E2). The explanations found in the cases were two: first, in two of the cases the systems had a common ancestry since previous collaborations as far back as twenty years or more (I_{Da}:1, I_{F2a}:1). Second, there 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 (I_{Cb}:1,7¸ I_{F2a}:1). This is also illustrated by the fact that all instances of similar architectures found were at the system level, in no case within a component in a decomposed system (although we do not think this would be impossible).

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

We have met the opinion from researchers and software practitioners that these observations confirm intuition, on the border to being trivial. On the other hand, we have also encountered contrary views mainly from management (for example in the cases), that it should be straightforward to reuse and merge the best parts (i.e. implementations) of the existing systems. Our contribution is thus the confirmation of knowledge that some call common sense, but others apparently are not fully aware of.

5 Conclusion

The topic of the present paper is how organizations in control over two or more similar software systems, typically as a result of a company merger or some other close collaboration, can arrive at a single software system. Let us recapitulate the two questions asked in the beginning:

Q1. Which are common experiences (good and bad) concerning reuse when merging two or more systems?

Q2. To what extent are the lessons learned from these experiences possible to generalize into recommendations for other organizations?

The experiences from the cases answer Q1, and can be used as a reference for how specific problems were solved in the past. The observations were structured into three

types of reuse (reuse from all, reuse from one, and no reuse) of four artefacts (requirements, architectural solutions, components, and source code), and some general patterns were described on when it seems appropriate to reuse.

Answering Q2 means motivating the external validity of the research. The cases all concern large-scale software systems with development and maintenance histories of many years or decades, and the wide range of domains represented hints at the observations being representative for a large number of integration and merge efforts.

One can argue that studying other cases in slightly other contexts would lead to different results. For example, companies using a product-line architecture approach, or COTS-based development might give different observations concerning reuse. There are also known patterns or best practices that are not directly in line with our observations. For example, the existence of architectural patterns that are reused in completely different systems with completely different requirements could be taken as a contradiction of observation 1 (which says that without reusing requirements, reuse of the architecture is difficult). While this can be true for functional requirements, it is not so with the non-functional requirements; in the case when we use particular architectural patterns we want to provide solutions related to specific concerns (for example reliability, robustness, or maintainability) that might not necessarily be explicitly specified as requirements.

Similarly, observation 2 (which says that the reuse of components is difficult or impossible, without reusing the architecture) may seem to contradict the component-based approach in which the systems are built from already existing components (i.e. components can be developed independently of the systems, and the component developers can completely be unaware of the systems which will use these components). However, it is known that with a component-based approach, an architectural framework is determined and many architectural decisions are assumed. Indeed, it is known that it is very difficult to reuse components that assume different architectural styles or frameworks. In this way, observation 2 confirms the experience from component-based approach.

Finally, we would like to comment on the remark that reuse of source code is opportune even if the components or architecture is not reused. This is of course true, although it is about reuse on a low level, which can very likely result in abuse (for example if the same source code is reused on several places without some synchronization mechanism). This observation might be an indication of a lower maturity of the development organisations, or for a need to encapsulate such code in components. This also suggests that when the new, integrated system is released the existing systems should be discontinued.

With this argumentation, we can claim a certain generality of our observations.

5.1 Future Work

Further cases could reveal more details and would also either support or contradict the observations presented in the present paper. Other case studies and theoretical reasoning could provide further advice on how systems built with different architectural styles and in different frameworks can, and should, be integrated.

To succeed with integration, not only technology is important. We have analyzed the case study material from a process point of view [15], and will continue by

describing how to choose between high-level strategies such as merging existing systems, discontinuing some systems and evolve others, starting a new development effort, or even doing nothing but let the existing systems live side by side. The selection among these strategies arguably involves much more than technical aspects; the observations on reuse in the present paper are only one among many influences.

A fundamental question is what *architectural compatibility* or *similarity* means in practice, and how dissimilarities can be overcome; the answer would presumably involve both what we have labelled "structure" (including e.g. architectural styles and patterns) and "framework". Even if the structure is dissimilar, the role of some components may be similar enough to allow for component reuse; the circumstances under which this could be possible need to be identified. One important issue for compatibility could be the notion of *crosscutting concerns* from the field of aspect-oriented programming [11]. The data models of the existing programs would also need to be taken into account, something we have only touched upon in the present paper.

There exist standardized and commercial solutions for integrating information systems (such as enterprise resource planning systems) that has been acquired and cannot be modified (only customized, wrapped, etc.). One question to study is how to choose between a tight merge and a loose integration in the context we have studied, i.e. when such systems have been developed and are fully controlled within a single organization. Such a study would presumably need to focus around data integration.

Acknowledgements

We would like to thank all interviewees and their organizations for sharing their experiences and allowing us to publish them.

References

1. Abowd G. D., Allen R., and Garlan D., "Using Style to Understand Descriptions of Software Architecture", In *Proceedings of The First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.
2. Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.
3. Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition* (20th Anniversary edition), ISBN 0201835959, Addison-Wesley Longman, 1995.
4. Bushmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, John Wiley & Sons, 1996.
5. Crnkovic I. and Larsson M., *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.
6. Fayad M. E., Hamu D. S., and Brugali D., "Enterprise frameworks characteristics, criteria, and challenges", In *Communications of the ACM*, volume 43, issue 10, pp. 39-46, 2000.
7. Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2, Addison-Wesley, 1995.

8. Garlan D., Allen R., and Ockerbloom J., "Architectural Mismatch: Why Reuse is so Hard", In *IEEE Software*, volume 12, issue 6, pp. 17-26, 1995.
9. Johnson R. E., "Frameworks = (Components + Patterns)", In *Communications of the ACM*, volume 40, issue 10, pp. 39-42, 1997.
10. Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.
11. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J.-M., and Irwin J., "Aspect-Oriented Programming", In *Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, Springer-Verlag, 1997.
12. Krueger C. W., "Software reuse", In *ACM Computing Surveys*, volume 24, issue 2, pp. 131-183, 1992.
13. Land R. and Crnkovic I., "Existing Approaches to Software Integration – and a Challenge for the Future", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, Linköping University, 2004.
14. Land R., Larsson S., and Crnkovic I., *Interviews on Software Integration*, report MRTC report ISSN 1404-3041 ISRN MDH-MRTC-177/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2005.
15. Land R., Larsson S., and Crnkovic I., "Processes Patterns for Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, 2005.
16. Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.
17. Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.
18. Schmidt D., Stal M., Rohnert H., and Buschmann F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, Wiley Series in Software Design Patterns, ISBN 0-471-60695-2, John Wiley & Sons Ltd., 2000.
19. Szyperski C., *Component Software - Beyond Object-Oriented Programming* (2nd edition), ISBN 0-201-74572-0, Addison-Wesley, 2002.
20. Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.
21. Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.