

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

**Techniques and Processes for
Assessing Compatibility of
Third-Party Software Components**

By
Laurens Blankers

Supervisors:

Michel Chaudron (TU/e)
Rikard Land (MDH)

Eindhoven, June 2006

Abstract

The increase in the use of software leads to a wish to exploit commonalities between them in order to achieve higher quality at lower cost faster. Like in other fields, the answer to this drive has been to create standardized components to fulfill these common needs. However the major difficulty with developing applications using third-party components is resolving the incompatibilities between them. The risks and costs associated with component based development can be reduced by thoroughly assessing components before integration in order to uncover architectural incompatibilities. Especially prototyping of combinations of components to test their interaction has proven to be useful in determining the suitability of components. Several techniques exist for assessing component compatibility based on properties of the component, however in order for these techniques to provide accurate results detailed architectural and design information is necessary, something which, in practise, is only rarely available.

Contents

Abstract	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Definitions	2
1.3 Contribution	3
1.4 Research Method	4
1.5 Outline	4
2 Survey on Assessing Compatibility	7
2.1 Introduction	7
2.2 Defining Compatibility	8
2.3 Detecting Mismatch	10
2.4 Resolving Conflicts	14
2.5 Discussion	16
2.6 Summary and Future work	17
3 Case Study on Component Assessment	19
3.1 Introduction	19
3.2 Research Method	20
3.3 The Cases	20
3.4 Method	23
3.5 Implementation	27
3.6 Related Work	29
3.7 Conclusion	29
4 Case Study on In-House Software Integration	33
4.1 Introduction	33
4.2 Research Methodology	34
4.3 Strategies	35
4.4 The Cases	36
4.5 Analysis	39
4.6 Related Work	46
4.7 Summary and Conclusions	48

5	Validation of Research	51
5.1	Introduction	51
5.2	Research Method	52
5.3	Goal	52
5.4	Design	53
5.5	Results	57
5.6	Conclusion	66
6	Conclusion	67
6.1	Future Work	68
	Bibliography	71
A	WICSA paper	77
B	Interviews	81
B.1	Case A	81
B.2	Case B	85
C	Questionnaire	89
D	Questionnaire results	105

List of Figures

2.1	Bipartite graph showing potential Problematic Architectural Interactions	11
2.2	Screenshot of Collidescope showing an architecture, component characteristics, and a PAI description	12
2.3	Ordering of types of control	13
2.4	Screenshot of Collidescope showing a solution.	16
3.1	Original Compatible COTS Components Selection Method	24
3.2	Extended Compatible COTS Components Selection Method	25
3.3	Flowchart of the PORE process	26
5.1	Current position	57
5.2	Company size in number of software developers	58
5.3	Domain	59
5.4	COTS usage	60
5.5	Requirements	60
5.6	Prototyping individual vs multiple components	61
5.7	Prototyping changes opinion	61
5.8	Available documentation	62
5.9	Initial information	63
6.1	Extended Compatible COTS Components Selection Method	68

List of Tables

4.1	Summary of the cases	36
4.2	The exclusion of possible strategies based on concerns	41
4.3	Concerns per case	42
4.4	Possible and desired strategies, per case	43

Chapter 1

Introduction

1.1 Motivation

Software is more and more common in daily lives today, ranging from embedded systems that control cars, airplanes, and even toys, to vast information system that control the always increasing amount of information, ranging from dictionaries, to stock markets, and medical systems. The increase in the use of software leads to a wish to exploit commonalities between them in order to achieve higher quality at lower cost faster. Like in other fields, the answer to this drive has been to create standardized components to fulfill these common needs.

The usage of third-party component in software development has sharply increased over the past decade [BB01] with the majority of software companies using third-party components to some extent, varying from one component which presents a limited part of the system to systems where over half of the functionality is provided by components.

With the increase of component usage both engineers and researchers became more aware of the problems associated with using third-party components. This led to increased awareness of software architecture. Many of the problems related to components have to do with the interaction between the component and the system and, in case multiple components are used, the interaction between components themselves. These conflicts were called architectural mismatch [GAO95] and the concept of architectural compatibility was born.

Architectural mismatches that remain undetected for a long time can reek havoc on software development, causing major delays, increased costs, and the possibility that the development will fail to meet its requirements. Since the cost and risks of any development increase sharply the further in the development process the project is, it is essential to assess architectural compatibility in the early stage of development [Som04].

In order to facilitate this, among others, two things are necessary. First of all a process for assessing components before they are integrated into a product. And second techniques to determine likely incompatibilities between components and between component and system at an early stage.

1.2 Definitions

Before discussing third-party components it is necessary to provide the definition of what we mean with third-party component.

There are many different definitions of what a component is. The SEI's COTS-based System Initiative defines a COTS product as [BOS00]:

- sold, leased, or licensed to the general public
- offered by a vendor trying to profit from it
- supported and evolved by the vendor, who retains the intellectual property rights
- available in multiple, identical copies
- used without source code modification.

This definition excludes open-source components (first bullet) and components developed by third-parties exclusively for a specific customer (fourth bullet). Szyperski defines a component as follows [Szy02]:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.

However components used in industry do not always have well defined interfaces [CL02]. The focus of this thesis is on the assessment of components for use in industrial projects. Since the properties of internally developed or in-house components are (supposed to be) known this thesis focusses on third-party software components.

For the purpose of this thesis we define third-party software component as follows:

A third-party software component is a self-contained piece of software developed by an external party.

This is a very broad definition that, we believe, covers most, existing definitions of third-party software components. In this definition the component needs to be self-contained, meaning that it should provide a defined function without the need for modification to achieve this function. Also the component should be developed by an external party, excluding in-house developed component of which the source code is available and can be modified, in which case the component is no longer self-contained. The definition does not require the component to be part of a component framework, or have a specific interface. In fact a database management system or operating system can be considered a component using this definition. Neither does the component have to be commercial or supported by one single vendor, open source components are also included in the definition.

The broad scope of this definition serves this thesis well because when it comes to assessing third-party components the importance of the assessment is related to the

importance of the functionality provided by the component to the end product. The importance is not necessarily related to the size of the component or whether it is commercial or open-source. Which means that for software components the issues are similar no matter whether the component in question is a small control routine for a mechanical device or a large distributed DBMS. The information necessary to assess the suitability and especially compatibility of a component are similar independent of the size or type of the component.

The assessment however is not limited to third-party components, since properties of in-house components are not always known, this is especially the case if a organization has multiple independent software organizations or in case the software components originate from two different companies in case of a recent merge. The thesis will focus on assessment of third-party components, however chapter 4 deals with the issues in merging in-house software systems after a company merger. This situation has properties similar to a component-based developed with third-party components especially when it comes to the familiarity of developers with the components.

From now on the term *component* will be used synonymous with *third-party component* as defined above, and similarly *compatibility* will be used in stead of *architectural compatibility*.

1.3 Contribution

Current research provides techniques for detecting and sometimes resolving incompatibilities between components on one hand and process framework for selecting suitable components on the other. However the proposed techniques are research prototypes and have not been tested in practise, while the processes are high-level and do not integrate with existing practise or with the proposed techniques.

This thesis aims to unify state-of-the-art research into both techniques and processes with current established practises in industry. Naturally the unification of two such vast areas, the academic research into component based software engineering on one hand and the accumulated experience and practise build up in industry on the other, cannot be simply unified in one thesis. However this thesis will hopefully contribute a small part to the unification.

To the unification this thesis contributes a state-of-the-art survey of techniques intended to determine the compatibility between components and between components and the system, including a discussion on the applicability of these techniques in practise. Furthermore an existing process model for component assessment is evaluated and extended based on experiences from two software engineers. Architectural compatibility is also viewed from a different context, namely that of a in-house integration in recently merged organizations. The contributions are not purely theoretical, rather they are based on experiences from practitioners obtained through extensive case studies and a questionnaire.

1.4 Research Method

Several methods and techniques are used in the research presented in this thesis. The specifics are described in the research method section of each chapter. This section will prevent an overview of the methods used.

The state-of-the-art described in chapter 2 is based on an extensive literature survey of professional literature including magazines, conference proceedings, theses, and technical reports. More details about the method used can be found in section 2.1.1.

Chapters 3 and 4 are based on multiple case studies [Yin03] through semi-structured interviews [Sea99]. In chapter 3 two extensive case studies were performed, while in chapter 4 consists of nine case studies. Details can be found in sections 3.2 and 4.2 respectively.

The validation described in chapter 5 was done through a survey in the form of an electronic questionnaire, more details in section 5.2.

Because of the use of case studies, not only the validation of the outcome has to be discussed, but also the validity of the case study research itself, because of different kinds of threats to its validity. These threats, and how they are addressed, based on advice given by Yin [Yin03], will be discussed here. The construct validity of the research is guaranteed by establishing a chain of evidence in the form of documented interview notes and the raw results of the questionnaire, both of which can be found in the appendices. Also the interview notes were presented to the interviewees for correction and feedback. Internal validity is addressed by separating the description of the data and its interpretation is much as possible. In order to establish external validity a questionnaire was created, which is described in chapter 5. All information necessary to repeat the study is included in this thesis, allowing for independent verification and reproduction of the research.

1.5 Outline

The remainder of the thesis, with the exception of the conclusion, have been written in the style of a series of scientific conference papers. Therefore each chapter is self-contained including introduction, research method, discussion, and conclusion. This allows for reading each chapter independent of the others. In order to facilitate the reader of the entire thesis each chapter is preceded by a short paragraph placing the paper in the context of the overall thesis. Included in this paragraph are details about contributions to the content by others and references to the conference where the paper was published, if applicable.

The thesis is organized as follows: chapter 2 contains a literature survey into the current state of the art of scientific methods for determining architectural compatibility between components. The survey is divided into four sections, the first part collects the different definitions of (architectural) compatibility, the second describes techniques for detecting mismatches between components that could negatively influence the compatibility, and

the third section describes techniques for solving the subset of mismatches that cause architectural problems called conflicts. The fourth part can be found in section 2.5 and contains a discussion about the encountered literature and its applicability to real-world challenges.

Chapter 3 presents a process for assessing the compatibility of one or more components. The process is an extension of an existing process based on two cases. The new process, dubbed eXtended COTS Component Compatibility Selection Method (xCCCSM), extends the original process with an additional step and several new transitions, plus detailed suggestions on implementing the two most important steps.

Chapter 4 is not about third-party components, rather it looks at architectural compatibility in a different context namely merging systems within a single organization. As described in the chapter, although a quite different domain from component based development, both have in common that compatibility has a major impact on the the possibilities, risks, and changes of success of a project.

The assumptions about existing techniques made in 2.5 and the process presented in chapter 3 remain unproven in practise until they are validated. The validation is described in chapter 5, which describes five hypothesis and a questionnaire as a tool for validation them. Naturally the results of the questionnaire are presented in section 5.5.

Bringing this all together is chapter 6 which states the conclusions of the thesis, based on the research described in the previous chapters.

The appendices contain the WICSA Work-in-Progress paper which is a derivative of chapter 4, the interview notes that form the bases of chapter 3, and the questionnaire (appendix C) and results (appendix D) that form the foundation of the validation in chapter 5.

Chapter 2

Survey on Assessing Compatibility

This chapter contains a state-of-the-art survey of academic methods of assessing component compatibility, or rather on methods of finding incompatibilities between components. The survey explores the available techniques and possible usage in processes discussed in the remainder of the thesis.

This chapter is based on a paper created as part of the university course Research Methodology for Computer Science and Engineering (CT3340) at Mälardalen University, Västerås, Sweden.

2.1 Introduction

Research has shown that architectural compatibility between components or between a component and the system is a major issue when developing software systems, causing large delays, reduced quality and functionality, or even failure to complete such systems. However much research remains before even the fundamental question can be properly answered: “What is compatibility?”, “How can compatibility or the absence thereof be detected?”, and “How can incompatibilities be solved?”. These are the questions this state-of-the-art literature survey was aimed to find the answers to.

In order to get a better insight in the issues surrounding compatibility, the ways of detecting them, and methods to solve them, the author has performed a literature survey into the subject. The survey tries to answer the following questions:

1. How is compatibility defined?
2. Which methods are available to determine the level of compatibility between components?
3. What methods exist to solve incompatibility?

4. How applicable are these methods in the context of a real-world organization attempting to integrate existing systems?

The survey follows the structure of the questions, with the result that sections 2.2, 2.3, and 2.4 will describe the answers current literature has to questions 1, 2, and 3 respectively. Section 2.5 will discuss question 4. Some approaches are mentioned multiple times in different sections because they provide answers to multiple questions.

2.1.1 Research Method

The survey that is the bases for this paper had the goal to find the state of the art regarding incompatibility, interoperability conflicts, and architectural mismatch on the abstraction level of the software architecture [BCK03]. This was done by screening relevant conferences, such as, but not limited to WICSA, ICSE, and ESEC/FSE in the past 5 years. From the articles that were deemed relevant the references were screened and also using Google Scholar [goo] and CiteSeer [cit] other papers that referred to the relevant paper were searched. Further more using the same services all references to, what is considered to be, the first article about this field [GAO95] were screened.

This method of screening assumes that all publishing researchers in this area are part of one or more networks that refer to each other's work and that some of this works gets presented at conferences in the field or published in well known publications.

The questions asked in the papers surveyed are characterisation and method questions, such as "Which characteristics influence mismatch?". Validation of the research presented in the surveyed papers is mainly through persuasion and in some cases through a prototype implementation.

2.2 Defining Compatibility

In the context of software systems, *compatibility* means the ability of different components to work together in a system that combines their functionality. Compatibility as such is a difficult concept to define, because it has very many aspects making it hard to say with confidence that two components are compatible. Rather researchers focus on *incompatibility*, a concept that is much easier to define since identifying one incompatible aspect leads to incompatibility. Because of this no researchers will claim that two components are compatible, rather they will claim that no incompatibilities have been detected, which does not mean that they do not exist. Incompatibilities can occur in many different areas including incompatibility of data, either in syntax or semantics, or incompatibility of environment such as programming language, operating system or hardware. These types of incompatibilities have been researched in the past resulting in a better understanding and some solutions.

2.2.1 Mismatch

This paper will focus on architectural compatibility, an area much less understood both in research and practise. Architectural incompatibilities are caused by assumptions components make about the way they interact with other components in the system. If these assumptions are not compatible this results in a mismatch. However a mismatch does not necessarily result in an incompatibility, although it is a strong indicator of one. This was first observed by David Garlan et al. in a ground breaking paper entitled “Architectural Mismatch or Why it’s hard to build systems out of existing parts” [GAO95].

These assumptions, when separated from the functionality of the component, are dubbed *packaging* by Shaw [Sha95]. In this paper she introduced the concept of connectors between components in order to be able to better understand the issues surrounding packaging. We will discuss connectors in more detail in section 2.4.2. One could ask why not build all component according to one packaging standard so that all assumptions are the same. Unfortunately, as the old quip goes, the best thing about standards is that there are so many to choose from. Also software architecture research indicates that a single standard will never be appropriate for all components [SG96].

2.2.2 PAI

Davis et al. refine the concept of mismatch by using the term Problematic Architectural Interaction, or PAI for short [DGP⁺01]. They define this as follows:

Definition Problematic Architectural Interaction. An interoperability conflict that is predicted through the comparison of architecture interaction characteristics and requires intervention via external services for its resolution.

The method distinguishes four different interaction assumptions ranging from Component-Component to Application-Application. Each interaction assumption has a number of characteristics that need to be compared to determine if any PAIs exist. Examples of such characteristics are: Blocking, Control Topology, and Synchronization. Values of characteristic do not necessarily have to be different to cause PAIs, identical values can also lead to problematic interactions. Also the values of combinations of characteristics can lead to PAIs even when the values of the individual characteristics would not.

2.2.3 Yakimovich

Yakimovich et al. have a similar definition, dividing problems into: syntax, first, second and N-order semantic-pragmatic incompatibility. Where a first order semantic problem is a problem with the component itself, e.g. component does not provide the required functionality, a second order describes problem between the interaction of two components. Incompatibilities caused by multiple components together are called N-order semantic-pragmatic incompatibility [YBB99]. Yakimovich et al. also distinguishes between different types of interaction. A component can interact with the platform, the hardware, the user, and other software [YTB99].

2.3 Detecting Mismatch

It is important to detect architectural mismatch as early as possible in the development process because the cost of changes to the design increases exponentially as the project progresses [Som04]. Methods for detecting mismatch can be classified into two broad categories, they are either *attribute based* or *model based*.

Attribute based techniques inspect the a set of critical assumptions or attributes of a component, such as control structure, layering, and concurrency, and the structure and connectors between components. These attributes are quite general and should be available for commercial components. Some methods are also able to handle the lack of information about certain properties. These methods requires only little information and time and therefore is well suited for evaluating several design alternatives early on in the development process. The disadvantage is that its mismatch detecting is pessimistic resulting in the detection of mismatches that might actually not occur; this is especially true when the values of certain attributes are unknown. A survey of the methods available follows in section 2.3.1.

The other method of detecting mismatches is based on a model of the system. Such a model can be either a description in a formal Architectural Description Language (ADL) or in a suitable process algebra. Modelling a system requires detailed knowledge about the system and is generally very time consuming. The advantage of this approach is that the resulting mismatches are more accurate. Because of the time required this method is best suited at the end of the architectural design phase to verify the decision for a specific architecture and to identify problematic mismatches that will need to be solved during implementation. A description of the methods available is presented in section 2.3.5.

2.3.1 Attribute

This section discusses three attribute based techniques: PAI, AAA, and Yakimovich.

2.3.2 PAI

The method identifies 13 PAIs divided into three categories: control transfer, data transfer, and interaction initialization. The notation used to denote PAIs is as follows:

$$C_i.s_r(Q) \rightarrow T \leftarrow C_j.s_t(R)$$

Where C_i and C_j are categories, s_r and s_t are value, Q and R are components, and T is a PAI. For example:

$$CS.Single - Thread(A) \rightarrow \{3\} \leftarrow Bk.Non - Blocking(B)$$

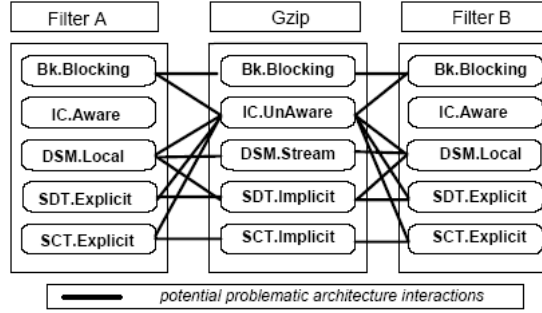


Figure 2.1: Bipartite graph showing potential Problematic Architectural Interactions [DGP+01]

indicates that component A has a single-threaded control structure while component B is non blocking, this causes PAI number 3 (Inhibited rendezvous). Davis et al. provide a formal definition of this notation [DGP+01], which is symmetric, however not reflective or transitive.

They introduce an union rule allowing the same value/component pair causing multiple PAIs to be combined:

$$C_i.s_r(Q) \rightarrow T \leftarrow C_j.s_t(R) \wedge C_i.s_r(Q) \rightarrow V \leftarrow C_j.s_t(R) \Leftrightarrow \\ C_i.s_r(Q) \rightarrow T \cap V \leftarrow C_j.s_t(R)$$

indicating that two characteristics cause multiple problems, where all could be solved if the values of the characteristics are adjusted. And an addition rule:

$$C_i.s_r(Q) \rightarrow T \leftarrow C_j.s_t(R) \wedge C_i.s_r(Q) \rightarrow R \leftarrow C_j.s_v(R) \Leftrightarrow \\ C_i.s_r(Q) \rightarrow T \leftarrow C_j.s_t(R), C_j.s_v(R)$$

which indicates that multiple characteristics can lead to the same problem.

PAI comes with an eight step method for gathering and solving the problematic interaction. This method includes creating a bipartite graph of the PAIs as shown in figure 2.1 and using the union and addition rule to reduce to number of PAIs. The 13 defined PAIs are described in more detail in a later publication [DFGK03].

This process is rather tedious to perform manually, therefore a prototype tool called Collidescope is provided to support this process [DG02] and can be found, free of charge, on the internet [col]. The tool allows designers to input an architecture using a graphical user interface, the tool will check this design for PAIs and also suggest a possible strategy to resolve the problems. See Figure 2.2. More about resolution can be found in section 2.4. The easy to use nature of the tool makes it also suitable for exploratory scenarios.

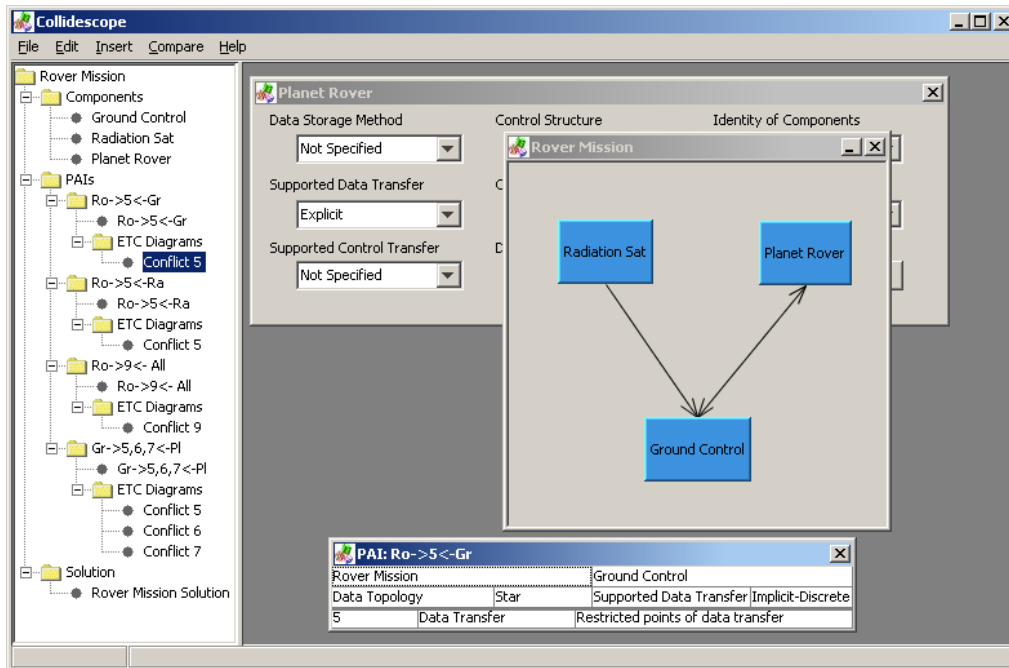


Figure 2.2: Screenshot of Collidescope showing an architecture, component characteristics, and a PAI description

2.3.3 AAA

The method proposed by Egved et al. is similar to the one described above, however their set of attributes is quite different [EG99]. The method also provides support for components that follow a certain architectural style, in that case style specific attributes are inherited from the style.

The authors also provide a prototype tool to support their process called AAA. Although it does not provide a GUI, it can handle seven different types of connectors as opposed to the two supported by Collidescope (uni-directional and bi-directional). The tool provides a comprehensive list of the detected mismatches [EMG00].

2.3.4 Yakimovich

A third approach by Yakimovich et al. does not consider attribute values as conflicting but rather as a partial ordering so that for a certain attribute value A is compatible with value B, but not the other way around [YBB99]. For example a multi-threaded component can call a component that have no concurrency or component that make no assumptions about concurrency, such as certain libraries, however the other way around is certainly more difficult, see figure 2.3. No detailed process or tool support is provided by the authors.

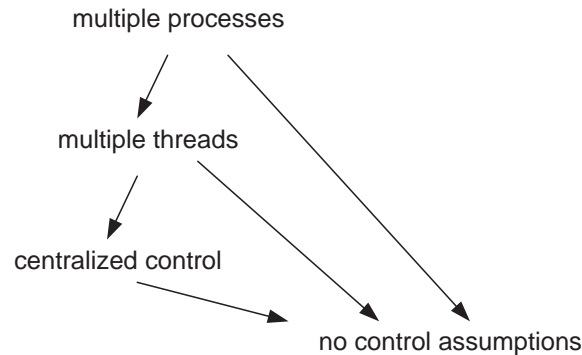


Figure 2.3: Ordering of types of control, a component can use another component at the same or lower level [YBB99]

2.3.5 Model

When the design of a system is finalised a detailed model of the system can be built using a formal modelling technique. The model can then be used by model checkers to determine whether it is complete and complies with the assumptions made. In order to be able to model a system detailed information about the design is necessary. A model is always an approximation of reality, in order to be manageable the modeler needs to use abstraction to model only the properties of the design that are relevant. Whether or not a design property is relevant is difficult to determine. An incorrect abstraction may make the model invalid.

Specifically for the purpose of modelling architectural design so called Architectural Design Languages have been developed. General ADLs such as Aesop [GAO88] and Wright [AG97] can be used to model both the system and the architectural style it needs to comply with. For certain architectural styles ADLs have been developed that are especially suited for that style. Layered systems can be modelled using the GenVoca ADL [BO92], event-based systems by Rapide [LV95], distributed systems using Darwin [MK96], and the C2 architectural style can be modelled using C2SADEL [MRT99]. These ADLs can be used in combination with automatic model checking tools such as, in the later case, SAAGE [TMA+96]. A description of the usage of C2SADEL and SAAGE on an example can be found in [EMG00].

Most mismatches occur in interaction between components, therefore process algebras can be used to model these interactions. Process algebras are mathematical languages designed to model and reason about processes. There are many different types of process algebra allowing modelling of different aspects and domains. For example certain process algebras can model timing making them suitable for modelling real-time systems. Other PAs abstract from timing making them suitable for situations in which the actions of the system are important rather than the timing. Research is ongoing in the area of using PA to detect architectural mismatches. Including work by Bernardo et al. [BCD01, BCD00] and by Inverard et al. who use the linking to chemical reaction in their Chemical Abstraction Machine (CHAM) to describe architectural interactions [IW95].

2.4 Resolving Conflicts

Once a mismatch is identified in a design it needs to be resolved. Resolution can be done in two ways. The first way is to identify the components that causes the mismatch and modify the design replacing one of them by a component that is compatible with other components and the system. Certain components, however, may not be replaceable for varying reasons. For example the offending component may be the legacy system in an integration project, in that case it replacing that component would require building the entire system from scratch which often is not a viable option. Another possibility is that the component in question provides functionality essential to the system and no other suitable component exists. In that case it may be possible to replace several other components or modify the architecture of the system to resolve, or at least reduce, the number of mismatches.

The second option to resolve mismatch is to use so called gluecode. Shaw was the first to notice that the problem with component mismatch often is not with the functionality of the component, but with the packaging of this functionality [Sha95]. Or in other words which type of connectors can be attached to the component.

2.4.1 Modify component

Possibly the first solution to come to mind is: if it doesn't fit, make it fit by modifying the source code. This strategy will only work in cases were the actual source code is available for modification, which is not the case for many commercial components. Even if the code is available it is sometimes questionable whether this approach is viable. Modifying existing code means investing time and money into the modification and requires retesting of the modified component. Studies show that extensively (> 25%) modifying the source code results in the same amount of errors as when building the component from scratch [Pou96]. This reduces the increase in efficiency that is normally associated with component reuse. In case of legacy systems it is also possible that there is insufficient knowledge of the legacy component to be able to modify it.

Some authors suggest that the packaging of a component should be separated from the functionality, allowing for easy adaption of the packaging to suite different architectures and connectors [DeL99a]. However this would require the components used to be written according to such a strategy, which unfortunately, for many is not the case.

2.4.2 Wrap

Wrapping involves using additional *gluecode* around the component in order to solve the architectural mismatch between the component and the rest of the system. In this approach the original component remains unmodified, however the overall system becomes more complex because of the added code. There are several approaches to implementing this strategy, which can be divided into three categories: adaptors, connectors, and middleware [MBF99].

An Adapter is a design pattern [GHJV95], also known as Wrapper, which acts as an intermediate between the component and the rest of the system in order to solve architectural mismatches. A mismatch that can be solved using an adapter is a type mismatch between two components. An adapter can translate one type to the other.

In the mid-90s Shaw started describing connectors between components as an equally important part of a system as the components themselves and elevating them to “first-class citizens” of software architecture [Sha93, Sha95]. Other researchers have identified connectors as important tools in connecting components in such a way as to resolve architectural mismatch. Connectors are not independent of other techniques of describing mismatch, in fact Adapters can very well be described in terms of connectors [DeL99b]

Deline describes eight connectors designed to resolve architectural mismatch [DeL99b]. These connectors are not newly invented concepts, but codify the experience of software architects in the same way patterns codify programming techniques. The list of eight connectors is by no means complete, neither is it intended to be. One year later a paper by Mehta et al. attempts to classify connectors in a similar way as biology classifies living organisms in order to get a better understanding about them [MMP00]. This classification framework contains eight main types of connectors with a total of 45 connector dimensions, which in turn have subdimensions. These subdimensions have a value. A connector is a combination of values of the subdomains of a type. This results, potentially, in a huge number of connectors, with the possibility of even more since the authors themselves do not consider their framework to be complete. In the framework connector types are first divided into different services. For example a procedure call is part of communication and coordination. A type is then further divided into dimensions. For a procedure call these include parameters, entry point, invocation, synchronicity, cardinality, and accessibility. A dimension can be divided into subdimensions, such as data transfer, semantics, return value, and invocation record in the case of the parameter dimension of the procedure call type. Finally (sub)domains have values. A type of connector (or species) is classified in one or more (sub)domains. For example coroutine is a communication and coordination service of type procedure call with multiple entry points.

2.4.3 Collidescope

Davis et al., who coined the term PAI, also introduced three integration elements [DG02]: a *translator*, which translates messages between component which expect incompatible formats, a *controller*, which coordinates the flow of information, and an *extender*, which adds missing functionality. In their tool Collidescope they introduce a number of concrete instantiations of these integration elements such as formatters, routers, buffers, etc. Based on the PAIs the tool suggests a number of integration elements to resolve the conflicts. An screenshot of Collidescope presenting a solution is shown in figure 2.4.

A category of solutions for mismatch is middleware. Middleware provides distributed computing services that facilitate interoperability between components [DGP⁺01]. Middleware however tries to solve the symptoms of architectural mismatch not the cause.

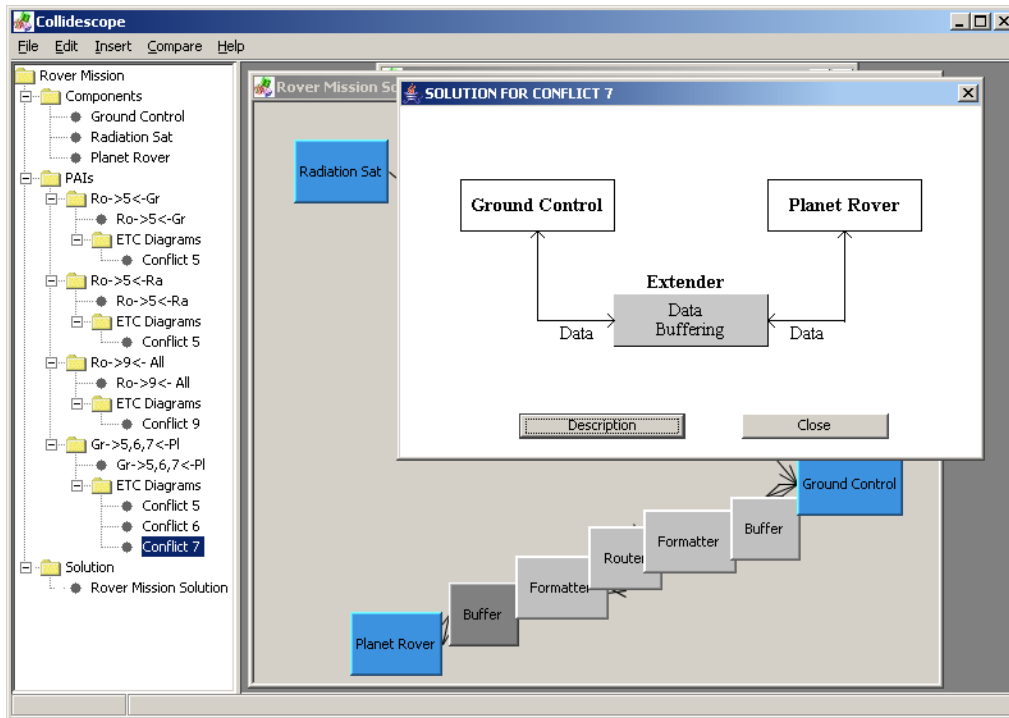


Figure 2.4: Screenshot of Collidescope showing a solution.

2.5 Discussion

The first three questions of this chapter (what is compatibility, how to detect mismatches, and how to resolve conflicts) that form the basis of this survey have been answered. The last question, whether this can be applied in an industrial context, cannot be answered based on the literature study because there simply is no literature available to answer it. None of the papers actually evaluate their proposed techniques in a real world environment, although some of the examples hint at being simplifications of real world problems. Not a single paper includes a reference to an actual evaluation. Therefore this section presents the author's opinion about the applicability in industry.

Research has made significant progress to answer the question "What is incompatibility?". A common understanding about the problem is required in order for a group of engineers to solve it. However the vocabulary used in the various research groups differs, making it harder to combine findings from multiple sources. Common terminology would make this easier, however as other fields have shown a single, universally agreed upon vocabulary is a utopia.

Several methods exist to assess incompatibilities in a design at different stages of the development process. Some of the methods provide tool support, making them easier and less time consuming to implement. Two of the attribute based detection methods provide decent prototype tooling, however they can only detect a limited number of mismatches and often the methods make pessimistic predictions, especially when presented with incomplete information, i.e. assuming mismatch where none exists. The model based

approaches provide more detailed and accurate information about mismatches, however it is very time consuming and different modelling languages, techniques, and tools are required for different architectural styles and problem domains. This makes these approaches suitable for domain specific development organisations in which correctness is critical (e.g. telecommunication), but makes them unsuited for software architecting in general. Hopefully the newly developed UML 2.0 [uml] will provide engineers with a single, standardised language, that will allow them to model every architecture. Providing adequate education and tooling is available.

As with any engineering discipline every problem is unique and requires an unique solution. That is not to say that solution patterns, as provided by some of the research described in section 2.4, are not useful. However since most of the solutions are formulated in terms of packaging and connectors that requires a paradigm shift from the developers building the components and the engineers attempting to apply these solutions.

In general several of the methods for both detecting and resolving mismatch are promising, however these need to address several issues in order to be useful to the general software architect. These include improving tool support, better handling of incomplete, and possible incorrect information, and standardization of models and notations.

2.6 Summary and Future work

The fact that architectural mismatches exist is widely known now, there also exist several classifications for the problems encountered, unfortunately none of them are widely accepted.

Methods to detect mismatch at different stages in the development process are being developed including the tools that make the assessments fast and simple. However none of the methods have been tested in the real world and all tools are still in the prototype stage.

Solutions to mismatches have start to appear described in the form of connectors, much like the object-oriented community describes design patterns. The connectors are not very specific, and much is still to be discovered about them.

2.6.1 Future work

As stated in the section on methodology none of the techniques described here, either for detection or resolution, have been evaluated in the real world. Or at least no scientific documentation is available describing such an evaluation. Therefore it would be of interest to try to use one or more of the tools and techniques described on an actual integration project in order to evaluate and build up experience with the proposed techniques.

Engineers in industry have been working on integration project for decades and, assuming that some of them were successful, must have found practical solutions to these

problems. Future work would include attempting to identify the engineers which such knowledge and encoding this knowledge into rules of thumb or systems in order to spread this information to the community at large and improve the overall quality of integration projects.

Chapter 3

Case Study on Component Assessment

Unlike the previous chapter, which deals with compatibility assessment in an academic context, this chapter deals current practises of assessing the compatibility of components in an industrial context. In this context compatibility does not only include technical compatibility between components, but also includes compatibility with the requirements and business.

3.1 Introduction

As more and more software COTS (Commercial Off-The-Shelf) components are available on the marketplace, organizations are trying to exploit the advantages these components provide by using them as part of their software products instead of developing the entire product in-house. Not all projects using COTS are a success though. A number of them suffer from problems with selecting the right components. Component may not fulfil the desired requirements, or the assumptions made in the component's design may not be compatible with the overall architecture of the system. There are several published methods and processes to evaluate software components and incompatibility. However not much literature exists about large scale assessment processes in industry.

In order to obtain this knowledge we performed two case studies at two large multinational companies, interviewing practitioners who had taken part in thorough assessment processes, which were considered successful by the people involved (meaning efficient and effective, i.e. the right decision was made). These cases are compared with two published assessment methods, of which one (by Bhuta and Boehm) fits the events of the cases and is further refined and extended.

The method of research is described in section 3.2. Section 3.3 describes the cases. The two published methods for the assessment process and their relation to the cases are discussed in section 3.4, including several extensions to the closest matching model. Recommendations on how to implement the information gathering steps is described in

section 3.5. Related work is discussed in section 3.6. Finally section 3.6 summarizes the main conclusions and outlines future work.

3.2 Research Method

We performed a multiple case study [Yin03], where we gathered information from two practitioners who were involved in a component assessment process. To collect the data, people willing to participate in the interviews were found through personal contacts. The interviews were held with one person in the each organization who had been directly involved throughout the whole assessment process. Both are experienced engineers and were partially independent because they came from outside the organization, in one case from the organization's research division in the other from a consulting company. The questions were open-ended, focused around technical aspects and processes. The interviews were recorded for the interviewer's personal use in copying out the interview notes, which were sent back to the interviewees for feedback and approval. The questions and answers can be found in the technical report.

The information gathered in the cases is independently confirmed by two experienced software consultants, referred to as C1 and C2, in the same respective domains as the cases. Therefore the extended method described here is not only based on two large cases but also on the accumulated experience of dozens of assessments.

3.3 The Cases

This section describes two cases of component assessment based on semi-structured interviews with two engineers each involved in one project.

3.3.1 Case A

Background. The company develops real-time safety critical embedded systems and is one of the largest in its domain. The interviewee is employed at the company's research division as a software engineer/researcher and was hired by one of its largest and most successful business units as the project leader of the assessment.

Situation. The component under consideration is a GUI builder that allows customers to customize the interface of a large system. The current system and components are working satisfactory, however the GUI builder was build on top of a platform no longer supported by the vendor. Therefore the goal of the project was to replace the component by one built on the new platform by the same vendor. Developers in the business unit preferred building the new GUI builder component in-house. Management, considering the advantages of COTS, preferred a commercial third-party component. An assessment project was initiated with the goal of determining whether an existing component or an in-house development would best meet the requirements and constraints, including not

only functionality and quality but also implementation characteristics such as short time-to-market.

Assessment. The project team included the interviewee as project leader, three researchers from the research division familiar with assessment processes, three developers from the business unit, and several other persons providing services. The first two months were spent on refining the requirements. The initial selection of components was based on information gathered from trade fairs by management. The next two months were spent on assessing the components, reducing the 9 potential components to two viable ones, plus the option to develop in-house. After the short list was created the team continued, without the three researchers, by building prototypes using the two components and also created a prototype for the in-house development. A complicating factor was that neither of the vendors had a finished component based on the new platform, but were still in the early development stages of creating one. The main problem encountered during the prototyping was the incompatibility between the hierarchical way the system was organized and the way the components were organized. However since both vendors were in a very early stage of development and both were willing to cooperate with the company there existed the possibility of a jointly developing a suitable component as a separate product line.

Decision. Both vendors were willing to accommodate the specific requirements by introducing a separate product line. However, the main question of the assessment team was whether this would be an advantage over building the component in-house. Because the vendors were in an early stage of development themselves a shorter time to market and other advantages associated with mature COTS components did not apply here. Furthermore an external component would mean that core knowledge and partial responsibility for customer support would lie outside the company. The team concluded that the best course was an in-house development of the component.

Evaluation. According to the interviewee the challenge in this assessment was not so much technical as it was to get a common understanding of the situation and the solution. Both in the technical aspect and in creating consensus the assessment was very successful. One of the main contributors to this success was the involvement of the developers from the business unit.

3.3.2 Case B

Background. The financial institution is among the largest in the world. It hired the interviewee from a large consulting company as part of a team of consultants specifically for the purpose of gathering requirements, assessing viable components, and guiding the integration thereof. The interviewee has more than 10 years experience in software architecture and design on numerous projects in this domain.

Situation. There were a large number of systems loosely integrated with information being both distributed and duplicated. In 1999 it was decided that, in order to better serve their customers it was desirable to have uniform, coherent information about customers throughout all systems. In order to achieve this, a uniform integration of

all systems was required, an integration which is nowadays referred to as an Enterprise Service Bus (ESB). The systems to be integrated communicated with each other in an ad-hoc manner, both in a synchronous and asynchronous fashion.

The base of an ESB is a message bus. Since the market for message busses was considered mature the financial institution wanted a COTS message bus component with additional services built on top.

Assessment. The assessment process was performed by five to ten consultants with experience in this area. The process started out with thorough requirements gathering process based on the input from the institution and the experience from the team members. The requirements were organized according to ISO 9126 [iso] and prioritized.

The initial selection of components was made mainly based on the knowledge of the team members about the available components. Some were immediately discarded because they were considered not mature enough. This resulted in a long list of about ten components. After comparing the product features with the requirements about five components were still under consideration. The vendors of these components were invited to give a presentation about their product providing more input to the assessment. Based on the requirements this resulted in a short list of three components.

The vendors on the short list were asked to fill out a detailed questionnaire with open questions based on the requirements. The answers to the questions were then graded by the assessment team. The grading of the answers was discussed with the vendors, to ensure correct interpretation. The scores assigned to the requirements together with the priorities resulted in a clear ranking for the three components. The team recommended the best of the components to the institution's management.

Decision. There were existing ties between the institution and the vendor of the component ranked second, and management considered this vendor more reliable (reliability being very important in the financial domain). The management therefore strongly urged the project team to reconsider the component ranked second. The assessment team accepted this component as a sufficient replacement, as it also fulfilled most requirements; the main difference between the two components being that number one supported synchronous communication natively, while the number two did not. A prototype was created for the now selected component to act as proof-of-concept. The prototype fulfilled most of the requirements and the decision was made to build a synchronous communication layer on top of the selected component.

Strictly speaking this ended the assessment process, however relevant for this paper is that the attempts to build the synchronous communication layer ran into difficulties and it was decided that this was not a viable option. Since the product had been selected it was decided to change the scope of the project to include asynchronous systems only. *Evaluation.* Although the project did not achieve its goal of providing an ESB for the entire company, but instead was only used for integrating backend systems, and although the costs were more than twice the budget, the assessment was considered a success, with much lessons learned for improvements in the future.

3.4 Method

Two published methods have been found which applies to the assessment processes in the cases, these are the Compatible COTS Component Selection Method or CCCSM [BB05] and Procurement-Oriented Requirements Engineering or PORE [NM01]. These are described below, with emphasis on how the model and observed process deviate. The description of the closest matching model, CCCSM, is then extended to better represent what happened in the cases and most likely better suite assessments in general.

3.4.1 CCCSM

The Compatible COTS Component Selection Method by Bhuta and Boehm [BB05] describes 11 steps and the transitions between them shown in figure 3.1 on page 24. Based on the events of the cases, several extensions to the model are proposed, depicted in figure 3.2 on page 25. The extensions to the original method are highlighted using thick boxes and arrows. The names in Figure 1 are the original ones, the ones listed in the following text are our own abbreviations.

Step 0: Identify OC&P. Both cases performed an extensive requirements gathering effort before proceeding to the identification of the candidate components.

Step 1: Identify candidate components. The identification was made using information gathered from trade fairs (case A), past experience, and publicly available information on the internet (case B). Both aimed for finding as many candidates as possible without regards for their viability at this stage.

Step 2: Classify into function groups. Both cases were assessing a single component, therefore step 2 is not applicable here.

Step 3&4: Evaluate alternatives & Gather information. The gathering of information for the evaluation was not done by buying information as proposed in the original method. In stead the vendors provided detailed, customized information to the assessment free of charge because of the attractiveness of acquiring a contract with the assessing organization. The way in which the information was solicited was different between the cases. In case A the bulk of the information was gathered during face-to-face meetings between developers from the organization and the vendor, this was called for because the vendors were still in the early development stages of their respective components.

Case B invited presentations/demonstrations from the vendors that were considered viable, which were followed by a detailed questionnaire, based on the requirements, send to the vendors that gave a good impression during the presentations. The answers to the questions were reviewed by project members and discussed with the vendors.

Additionally case A asked the vendor to fill out a standardized supplier questionnaire enquiring about the financial stability of the vendor. Case B requested

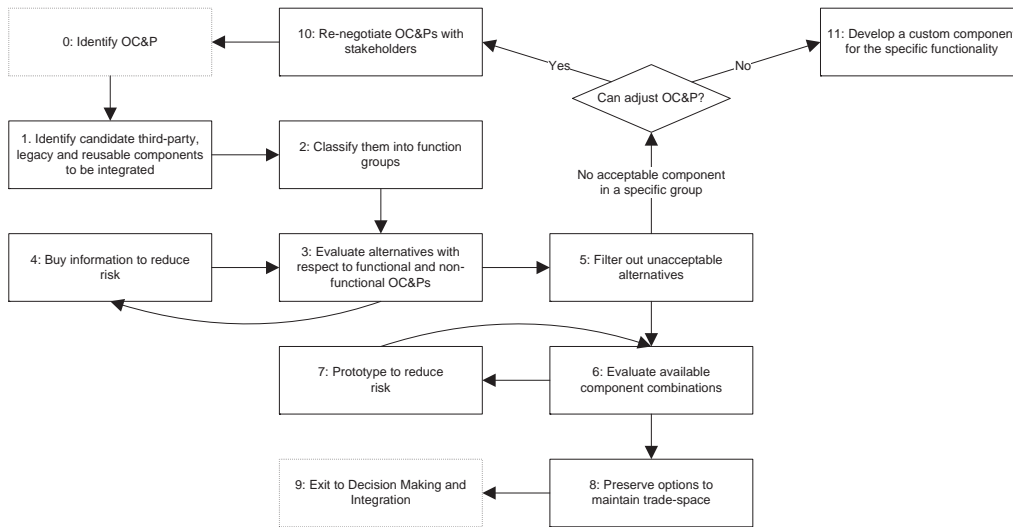


Figure 3.1: Original Compatible COTS Components Selection Method (CCCSM) [BB05]

references from the vendors on the short list and interviewed them about extra-functional properties such as reliability. Discussions with independent consultant C1 in a large company in the same domain as case A confirms that vendors are willing to provide the requested information, although some react surprised to these requests because other customers tend to not request such information.

Step 5: Filter out alternatives. This step resulted in two viable components for case A and a short list with three potential candidates (with a clear preference for one) for case B. All other components were discarded because they did not meet the requirements, either functional or extra-functional such as maturity and reliability.

At this stage the case B management intervened and expressed their strong preference for the number two on the short list. Subsequently the preferred choice was changed.

Step 6&7: Evaluate & Prototype. Additional information for the evaluation step was obtained through prototyping in both cases. However in case A for both viable components as well as for an in-house development a prototype was created. In case B only the selected component was prototyped. In case A the prototyping of both components led to the conclusion that neither would meet the requirements, especially those regarding interacting with the existing system, because the structure was too different.

The prototype in case B was a success confirming that the component could fulfil the requirements that it had expected too fulfil, which does not include the native support for synchronous communication.

Step 6b: Filter out alternatives. This step is an extension to the original model. This step takes as input the results from the evaluation in step 6 and determines whether or not one or more viable components exist. This step is similar to step 5 which filters components based on the evaluation of gathered information (step 3 & 4). If one or more viable components are found the model continues to step

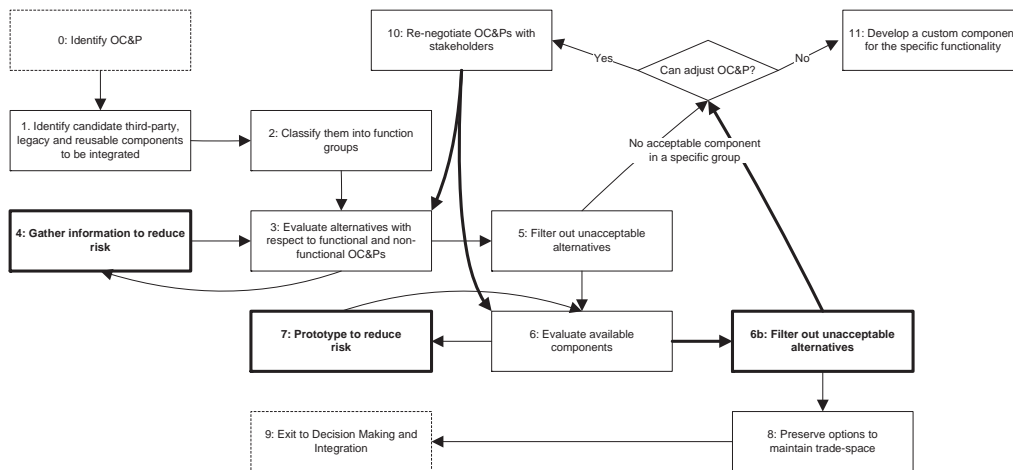


Figure 3.2: Extended Compatible COTS Components Selection Method (xCCCSM)

8 (Preserve), as it would have done in the original model. However if no viable components are found the assessing organization should ask itself whether the requirements (OC&P) can be adjusted.

In case A the results of the prototypes and evaluation thereof clearly conclude that none of the components is acceptable and thereby following the path to the question whether requirements can be adjusted.

In case B the decision made here is less clear. The component was selected with the knowledge that it would not be able to fulfill the requirement of being able to handle synchronous communication without an additional layer to manage this. Parts of the assessment team were skeptical about building such a layer. However it was decided to continue with this component and the additional layer. So strictly linearly speaking the decision in step 6b was that there was one viable component. However during the assessment and the attempt to build the layer that followed it was realized that the layer would not be able to fulfill the synchronous requirements and the decision in this step was revisited and changed towards considering whether the requirements could be changed.

Step 8&9: Preserve option & Decision making. Neither case explicitly preserved the information from the process to facilitate future changes. In case A preserving the information may not have been useful because of the rapidly changing market situation.

Decision: Can adjust OC&P? This question was asked in both cases once it was realized that none of the components were viable (as a result from step 6b).

In case A the requirements could not be changed because this would mean changing the existing system the component had to integrate in which was explicitly not allowed. Therefore the decision was made to build the component themselves.

The question of whether the requirements could be changed was raised in case B after the decision to select the component was made because step 6b was revisited. At this point it was undesirable to start a custom development and it was possible to change the scope of the project to only include asynchronous systems.

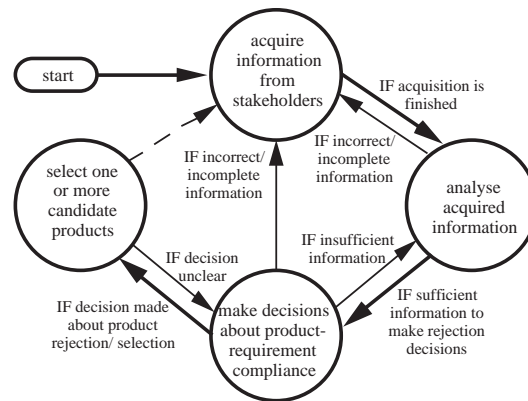


Figure 3.3: Flowchart of the PORE process [NM99]

Step 10: Renegotiate requirements. This step was taken in case B when it was decided that the requirements needed to be changed when the decision to select a component was revisited. The negotiations let to a reduction in the scope of the project so that the requirement that the ESB should handle synchronous communications could be dropped.

Unlike the original model the process did not return to step 0, in stead the process returned to the evaluation step (step 6) preceding the filter step (step 6b) that let to the exclusion of all components. Because of the changed requirements the component did fulfill all requirements now and the decision to select it was reconfirmed.

From this it logically follows that if this step had been reached from the first filter step (step 5) the process would have returned to the evaluation in step 3 without re-identifying candidate components and classifying them. Although it might be desirable to re-identify candidate component when the requirements have undergone a major change the authors believe that in actual processes this likelihood of this happening would be small.

Step 11: Develop a custom component. Since no viable components were found in case A the only option left open was to start an in-house development of the component with the desired functionality.

3.4.2 PORE

The PORE (Procurement-Oriented Requirements Engineering) method [NM01] describes an iterative process in which initial requirements are defined, components are evaluated and filtered based on these requirements, and the remaining candidates are explored to find new requirements. This process starts out with a number of high-level requirements and numerous potential components. After each iteration the number of components is reduced and the depth of the requirements is increased, till finally this results in an extensive requirements description and a single (or possibly a few) components that comply with these requirements.

Neither of the cases has defined their requirements in the iterative way described by the PORE method, in stead requirements engineering process was more like described in the waterfall model. However both cases did evaluate the potential components in an iterative manner. Starting with the highest priority requirements and the basic information provided by the vendor unsuitable components were removed early (case A) or were considered but did not make it onto the long list (case B).

Especially in case B this iterative process continued by inviting vendors to present their component, removing several from consideration, requesting vendors to fill out detailed questionnaires based on the requirements, removing even more, and discussing the answers with the vendors resulting in a short list containing three components with a clear number one.

3.5 Implementation

The cases suggest implementing several of the steps differently from the way proposed in the model. In particular step 4 and 7 are handled quite differently.

3.5.1 Step 4: Acquire information

In our extended model we rename step 4 to gather information in stead of buying it. We suggest that organizations try to request as much information from the vendor as possible, and use their leverage over the vendor (if any) to obtain this information. In case A the direct access to developers, and possibility of a cooperative development effort, allowed for a very direct and detailed exchange of information.

Furthermore requesting interviewing references is a good way of ascertaining difficult to assess properties such as reliability and responsiveness. Although it should be noted that vendors will only provide references with positive experiences and that references have a limited amount of time available to answer questions. Finally, if the assessing organization has a procedure for assessing suppliers in general this might very well apply to software component vendors also.

The fact that the cases had this extend of free access to information may very well be due to the fact that they were potentially very lucrative customers to the component vendors. In other circumstances, where the assessing organization is smaller, vendors might not be as inclined to provide this kind of information free of charge. The willingness to provide information may also depend on the payment terms. If payment depends on customizations made to the component by the vendor the vendor might be more willing to provide information than if the payment is done once or is based on royalties.

This is confirmed by an informal discussion with independent consultant C1 working for a large company in the same domain as case A, which describes the reaction of the vendor to their requests for detailed information as surprised but willing. Surprised because no other customer has ever requested that extend of detailed information, willing because of

the size of the company and willing because the company in question is a very lucrative customer.

In both cases the interviewees were asked how accurate the information provided by the vendor during this step turned out to be. Both confirmed that the information provided turned out to be accurate and complete. Neither considered the possibility that the information provided by the vendor was anything but accurate and complete.

In the assessment stage of case B vendor developers were not directly involved. However the interviewee describes the information provided by the vendor as part of a contract like construction in which promises made should be upheld by the vendor if his component was selected.

The trust the interviewees placed in the component vendors turned out to be well placed. However it should be noted that there is not guarantee that information provided by a vendor is both complete and accurate. Besides the attribute and the value also the credentials should be considered. The information provided might not always be accurate or complete therefore for every property the credentials [Sha96] should be considered. For important properties the values should be verified, not just taken on faith from the vendor.

Case A treats this assessment as any other assessment of a product offered by a supplier and therefore the company's standard procedure for assessing the financial stability of the suppliers is performed.

Both interviewees also mention that past experience either with assessments (case A) or with the components being assessed (case B) is an important aspect in the assessment and that the assessment team is constructed to include members with this experience. When assessments are performed regularly within the same domain it may be beneficial to determine common problems with component integration and document these for use in future assessment. This could take the form of a checklist containing categorized and prioritized questions which should be answered in order to assess a specific risk [TS06].

3.5.2 Step 7: Prototype

Both cases show that prototyping is essential to assessing components. The cases also show that prototyping is not necessarily an activity employed by the assessing organization alone.

In case A the developers had direct access to the unreleased version of the components and their developers for building the prototype. In case B several experienced integration engineers from the vendor were hired during the construction of the prototype. Unlike case A this service was paid for.

Having close cooperation with engineers from the vendors is not uncommon within this type of large integration projects. Informal discussions with an independent consultant C2 working in the same domain as case B indicates that vendors may even invite assessing

companies to their labs to build a prototype with assistance from the vendor's engineers free of charge if the potential benefits for the vendor are large enough.

This leads us to the conclusion that prototyping is essential in assessing a component and that vendors tend to be willing to invest in assisting the development of such a prototype providing the potential sale is lucrative enough. And even if they do not, hiring vendor's engineers for assistance in the prototyping could improve the quality of the prototype and thus the quality of the assessment greatly.

3.6 Related Work

There is a maturing component-based software engineering discipline, and although much literature and research focus the technical foundations [HCS⁺05, Szy02], many textbooks discuss both technical and managerial issues [HC01, Sam97, WHS01a] and/or include industrial case studies putting component-based systems in a larger context [CL02]. Only little concrete and validated research can be found on component-based processes; typically such literature describe at a fairly high level the difference between, and non-trivial combination of, processes for building components and building with components [CC05], or there are very detailed methods for particular aspects of assessment, such as for reliability [SSC04], without a larger process context. However, there is a noticeable lack of concrete advices concerning the crucial task of component selection and requirements engineering when building systems with third-party components. As described earlier in the paper, we have found two such suggested processes, the Compatible COTS Component Selection Method by Bhuta and Boehm [BB05] and the PORE (Procurement- Oriented Requirements Engineering) method [NM01]. The first one, that fits the cases presented in the present paper best, has not been independently validated before but should be seen as a suggestion based on experience. The PORE method has been applied rather successfully in at least two reported cases.

In order for a component to be successfully integrated it needs to fulfil the functional requirements. However it also needs to be compatible with the system architecture. A seminal paper published in 1995, first identified and labelled the problem of architectural mismatch [GAO95], resulting in integration problems. Among the publications that followed were papers detailing methods for assessing compatibility between components. These methods can be divided into two categories. The first uses basic attributes of the components, such as control structure and data topology, to determine potential mismatches quickly, examples are [DGP⁺01, EG99, YBB99]. The other category includes more formal modelling approaches, in which components are modelled in order to detect mismatches, these methods are more time consuming, but also more precise, and include the use of process algebras [BCD01] and ADLs [EMG00].

3.7 Conclusion

Based on the information gathered we conclude that using the extended CCCSM (x-CCSM) model in combination with the incremental assessment of components, which is

part of PORE, leads to a thorough component assessment process with a high likelihood of selection the most appropriate component. The original CCCSM method can be used to describe the process in both cases well. However it needs to be extended in some areas: adding a prototype evaluation step and shorter iterations after requirements have changed. Based on the cases we also provided recommendations for implementing the two assessing steps.

Companies gather information in a number of ways where buying it, as suggested by the original model, might be one. Other ways include requesting more information from the vendor, or interviewing references. Both the original method and the cases emphasize the importance of prototyping. However, unlike the original method, in the cases the prototyping was actively supported by the vendors. Moreover prototyping could result in none of the components being considered viable, if this happens the assessing organization should take similar steps as when no viable components are found after the initial information gathering step: The organization then has to consider whether the requirements for the component can be changed, and either change them or develop a custom component. After changing the requirements it is not likely that an organization will again assess the available components on the market. In stead it is more likely that it will return to the assessment step which led to the exclusion of all alternatives.

The PORE method proposes an incremental method of obtaining requirements, selecting components, and then refining the requirements and selection. Although neither of the cases obtained the requirements incrementally, both assessed components in an incremental way in the evaluation phase. Incrementally gathering requirements might however be very appropriate for smaller organizations that do not have the resources to build the component in-house if none of the available components exactly matches the requirements.

The details of the information gathering and the support from the vendors might be different depending on the size of the company. Smaller companies may not have the resources and/or get the support from the vendors to perform a very detailed assessment of each component. However as one of the interviewees mentioned, a less thorough though complete assessment does not necessarily lead to a lesser result.

The interviews also quite clearly indicate that an assessment process is not purely technical, but that a range of political, strategic, and organisational factors affect it. In case A the primary goal of the assessment was to independently determine whether the component should be developed in-house, as preferred by the developers, or bought from a third-party, as preferred by management. In case B management was instrumental in the final selection of the component, a decision that had significant impact on the scope of the project. Engineers involved in an assessment process should not underestimate the importance of these non-technical factors on the direction and outcome of the assessment.

3.7.1 Future Work

Validating whether xCCCSM is beneficial can be done in two ways. One is to systematically apply the method in a number of assessments and evaluating the outcome.

However we do not have the necessary resources to perform this type of validation. The other way is to extend this case study to more cases. For that purpose we are currently developing a questionnaire to be distributed to a large number of companies to ascertain whether their way of working matches that of the cases and whether their results and experiences are similar.

3.7.2 Acknowledgement

The authors like to thank the interviewees for openly sharing their experiences, who we, unfortunately, can not name them out of consideration for the companies they work for. We would also like to thank Eoin Woods for informal discussions which provided valuable insights, and also for his feedback on this paper, and Tim Trew for sharing his experience from numerous assessments.

Chapter 4

Case Study on In-House Software Integration

This chapter deals with merging two or more software systems with overlapping functionality within a single organisation. Although the original, unmerged, systems (or parts thereof) can usually not be considered components, the context of the merge exhibits significant similarities with component-based development, especially when it comes to the need for assessing compatibility. The situation, however, has several unique challenges one of which concerns the ‘retireability’ of one or more of the existing systems.

This chapter is based on the conference paper entitled “Software Systems In-House Integration Strategies: Merge or Retire - Experiences from Industry” written by Rikard Land in cooperation with the author of the thesis. The paper was presented at the Fifth Conference on Software Engineering Research and Practice in Sweden, Västerås, 2005 and also published as a short paper at the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), Pittsburgh, 2005.

4.1 Introduction

Organizations have spent large sums of money on development of software systems as part of their core business and want to capitalize on their investment. At the same time reorganizations and mergers force the organizations to integrate their software systems. This leads to a variety of problems such as functional overlap and architectural and platform incompatibility.

The software may be the core products of the companies, or a support systems for the core business. If 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.

The available published experience on integration does not directly address this context, where the existing systems are developed separately but now completely controlled within a single organization. We have chosen to label integration in this context in-house integration in contrast to other types of integration found in literature: integrating third-party components or systems, and interoperability based on standards in open systems. These approaches might be applicable also in the situation when an organization has full control over the development and evolution of the systems to integrate, but there are other options as well, such as modifying arbitrary parts of the existing systems in order to be able to merge them, starting a new development effort based on the previous experience, or selecting one of the existing systems and evolve it in order to replace all existing systems. To collect all the existing experience from organizations that have faced this challenge we have performed a multiple case study, consisting of nine integration projects in six organizations.

Based on the case study we present four integration strategies and two concerns that are crucial to address when selecting a strategy, which we have labelled compatibility and (lacking a better term) retireability. The audience this paper aims for consists of both researchers and industrial architects.

The paper is organized as follows: section 4.2 describes the methodology. Section 4.3 presents a model of four integration strategies which is used to classify the cases, which are described in section 4.4. The cases are further analysed in section 4.5 in terms of strategies and concerns. Section 4.6 presents related work, and section 4.7 concludes the paper.

4.2 Research Methodology

The multiple case study [Yin03] consists of nine cases from six organizations that have gone through an integration process. Our main data source has been interviews. To collect the data, people willing to participate in the interviews were found through personal contacts. The interviews were to be held with persons in the organization who:

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

All interviewees fulfilled either criteria 1 and 2 (project leaders with less insight into technology), or 1 and 3 (technical experts with less insight into the decisions made). In all cases, people and documentation complemented each other so that all three criteria are satisfactory fulfilled. There are guidelines on how to carry out interviews in order to e.g. not asking leading questions [Rob02], which we have strived to follow. The

questions were open-ended, focused around architecture and processes, and the copied out interview notes were sent back to the interviewees for feedback and approval. The interviewees were asked to describe their experiences in their own words, as opposed to answering questions, however we used a set of open-ended questions, to ensure we got information about the history of the systems, problems, solutions, and more. The questions are reprinted in the Appendix. Due to space limitations the answers are not reprinted. They can be found however in a technical report [LLC], together with further details regarding the research design. 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 (F1) one of the authors (R.L.) participated as an active member during two periods (fall 2002 and winter 2004-2005).

The research can be considered to be grounded theory [SC98] in the sense that we collected data to build models for previously un-researched questions, as contrasted to validating a pre-defined hypothesis. Strictly, there is no external validity in the traditional sense (of course the data fits the model, because the model is built from the data); validation would mean repeating the case study and comparing the current model with the new data. On the other hand, the new data would probably be used to modify or refine the model, leading to the same validity problem. It has been argued that for grounded theory research, the validation that can be achieved is a proper understanding, which can only be judged by other researchers [Max92, SC98].

We have deliberately avoided commenting 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? Or is the competitiveness and financial situation of the company a certain number of years a more interesting measure? And by making case studies, 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 opinions based on their perception of e.g. whether time and money was gained or spoiled.

4.3 Strategies

In order to classify the decisions made in the cases we present a model dividing the numerous possibilities for integration into four strategies that are easily understood analytically: No Integration, Start from Scratch, Choose One, and Merge. In reality, we can expect that these strategies are not strictly independent; some real cases can be seen as a combination of two, or as something in between. However we find it useful to use this model as a framework for discussion. The strategies, especially the Merge strategy, are discussed in more detail in section 4.5.1.

No Integration (NI) Develop existing software systems in parallel, which clearly will not result in an integrated or common system. However it is mentioned for the sake of completeness.

	Organization	System Domain	Goal	Information Resources
A	Newly merged international company	Safety-critical systems with embedded software	New HMI platform to be used for many products	<i>Interview:</i> 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	<i>Interview:</i> experienced manager and developer (I _B)
C	Newly merged international company	Safety-critical systems with embedded software	Rationalizing two core products into one	<i>Interviews:</i> 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 for Data-Intensive Server	<i>Interviews:</i> 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	<i>Interview:</i> project leader and main interface developer (I _{E1}) <i>Document:</i> 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	<i>Interview:</i> project leader and developer (I _{E2}) <i>Documents:</i> 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	<i>Participation:</i> 2002 (R.L.) (P _{F1a}); currently (R.L.) (P _{F1b}). <i>Interviews:</i> architects/developers (I _{F1a} , I _{F1b}); QA responsible (I _{F1c}) <i>Documentation:</i> research papers (D _{F1a}); project documentation (D _{F1b})
F2	Newly merged international company	Off-line physics simulation	Improving the current state at two sites	<i>Interviews:</i> software engineers (I _{F2a} , I _{F2b} , I _{F2c}); project manager (I _{F2d}); physics experts (I _{F2e} , I _{F2f})
F3	Newly merged international company	Software issue reporting	Possible rationalization by using one single system	<i>Interview:</i> project leader and main implementer (I _{F3}) <i>Documentation:</i> miscellaneous related (D _{F3a} , D _{F3b})

Table 4.1: Summary of the cases

Start from Scratch (SFS) 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 buying a commercial solution or building the new system.

Choose One (CO) 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 (M) 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, a very idealized strategy and as it will turn out the most complicated and broad strategy of the model.

4.4 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 demands on extra-functional requirements 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 4.1. They are labelled A, B, etc. Cases E1, E2, F1, F2, and F3 occurred within the same organizations (E and F). This paper focuses on architectural issues; for processes the reader is referred to [LLC05].

The remainder of this section describes the cases in some more detail illustrating how strategies were considered and how one was chosen. Due to space limitations this is done in a very brief manner.

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, that is, there was a wish to discontinue at least all but one of the existing HMIs. The two existing systems with the highest influence had a very different underlying platform: one was based on open source platforms and the other on commercial solutions. The differences were maybe largest when it came to the cultural clash associated with the platforms excluding the possibility of a Merge. 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. Therefore, Start from Scratch was desired by the architects over Choose One and thus selected.

Case B Two existing systems with similar functionality had to be merged in order to reduce cost. One system was used throughout the company the other only in one daughter-company. Discontinuing of the large system was not considered thereby excluding Start from Scratch. The smaller system was built on the tight integration paradigm, while the large system was build as a loose integration of many subsystems. The difference in approach made Merging the systems infeasible, therefore the Choose One strategy was chosen.

Case C The systems and the development staff of the organization is the largest among the cases: several MLOC and hundreds of developers. Two such systems with very similar functionality were being developed within the now merged company and both were nearing release. Management did not want to retire either of them, but wanted the best parts of both systems to form the new system within six months, i.e. Merge. The systems were similar in many ways, 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. The differences prevented Merging the systems in a short period of time, but allowed for Merging over a longer period. The architects reported to management that the Merge would take 2 years, much longer than desired, and that a better option was to Choose One. Eventually management changed its position so that either (but not both) could be discontinued, allowing for the Choose One strategy which was implemented. However the decision was not made until both systems were independently released and deployed at customers. This caused an estimated loss of one year of development effort of a team of several hundred developers, confusion to the customers who did not know which of the two products to choose, and

required addition effort in migrating the customers of the retired system to the chosen one. One of the interviewees points out that although the process seems suboptimal it is difficult to say whether other approaches would have been more successful.

Case D The two systems, both consisting of an HMI and a server, have a common ancestry, but have evolved independently for 20 years. Five years before the merger the HMI of one of the systems was decomposed into components and significantly modernized. The other system built on more aged technologies, and around the time of the merger the customers of this system considered that HMI to be outdated; it was therefore decided that the dated HMI should be replaced by the modern one, thus Choose One. In order to do this the server that used to be controlled by the dated HMI had to be changed, but thanks to the common ancestry it was relatively easy to make the same modifications to the server that had been made five years ago when the modern HMI was developed. The servers themselves have not been integrated yet; they both implement the same industry standards and the plans are to perform a gradual Merge but there are no concrete plans. In the summary and analysis of the cases we will therefore discuss the integration of HMIs and servers separately.

Case E1 The goal in this cooperation project was not only to integrate several existing systems, but also to add another, higher level of functionality. Retiring the existing systems was possible since all parties would benefit from the new system. Many of the existing systems (but not all) were written in the same language. However, a new language was considered better suited for a higher level of system complexity and would also bring a number of additional benefits such as reusable code, robustness, commonality within the organization. Thus Start from Scratch strategy was chosen.

Case E2 A certain functional overlap among three simulation systems was identified. 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. Two systems were somewhat compatible, but due to limited resources the only integration has been reuse of the graphical user interface of one into the other, although this was more complicated than anticipated. 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, the third system has in practice been retired and we consider this case to be a Choose One strategy (actually Choose Two out of three).

Case F1 After the company merger, there has been a need to improve and consolidate management and support for certain physics simulations with data management mechanisms and user interfaces. Initially, three systems were considered for integration or replacement; the two possibilities outlined were a tight merge with a result somewhere between Merge and Choose One and a loose integration in an Merge manner, which became the decision. However, the many incompatibilities indicated a very long development time. It was not considered possible to discontinue development on the existing systems before a full replacement was

available and the limited resources and other local priorities in practice resulted in no progress towards a common system. Currently, stakeholders are favouring Choose One, but the scope is unclear and integration activities still have a low priority; some participants have seriously begun to question the value of integration altogether and the result so far, after four years, has been No Integration. This apparent lack of results is not due to lack of will or effort, throughout these years there have been numerous attempts to identify a proper integration strategy.

Case F2 The two systems both consist of four programs run in sequence, with very similar roles, communicating via input and output files: pre-processor, 2D simulator, post-processor, and 3D simulator. 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. The 3D simulator is considered very large and complex to replace, so this is not realistic within the next few years. So far there are common pre- and post-processors; they have been rewritten, i.e. Start from Scratch, although the post-processor started as an attempt to Choose One of the post-processors and evolve it, but due to insufficient analysis of requirements it had to be almost completely rewritten. Although the 2D simulators are branched from a common ancestor, they are no longer very similar, and one of the 2D simulators is currently being evolved to replace the other, i.e. Choose One. Parts of the 3D simulators have been re-developed commonly, i.e. Merge. Due to the different states and choices of integration of the four parts we will treat them separately.

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. The two systems developed in-house were somewhat compatible. 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. Being a mature domain, outside of the company's core business, and realizing the effort required to creating a new issue tracking system from scratch themselves, the decision was to Start from Scratch by acquiring a commercial system.

4.5 Analysis

Many aspects affect the choice of strategy, which will be discussed in this section.

4.5.1 Refining the Model

In section 3 we presented a model to assist us in the discussion of the cases. Based on the cases we extend the Merge strategy, introduce two concerns that aid exclusion of strategies, and discuss influences on the selection of a final strategy.

Subdividing the Merge Strategy

To aid the discussion, we first present two types of Merge, labelled Instant 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. These two types of Merge should not be understood as two strategies with distinct identities in the same sense as the four originally presented. Instant Merge (IM) With “instant” 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. This strategy was desired in case C, but could not be implemented.

Evolutionary Merge (EM) Continue development of all existing systems towards a state in which architecture and most components are identical or compatible, in order to allow for Instant Merge sometime in the future. The architects in case C asserted that if a Merge was desired an Evolutionary Merge was the only possibility. Case F2 clearly demonstrates this strategy. Indications are that case D_{Server} will also follow this path in the future.

Concerns

Strategy selection in the cases was influenced by many factors of many different kinds, including the current state of the existing systems, both technically and from a management perspective, both in themselves and in relation to each other; 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; development resources; desired time to market; the impact of retiring any or all of the systems. To suggest a systematic procedure for selecting a strategy, one starting point would be to identify issues that would not only suggest one or more strategies as appropriate, but also exclude some strategies as inappropriate. Of the concerns listed, we identified only two of these as being able to, when properly addressed, exclude strategies: the architectural compatibility of the systems and the retireability of the systems.

Compatibility Architectural mismatch can make integration hard if not impossible [GAJ95]. In order for (parts of) systems to be integrated they therefore need to be compatible to some extent. Also when systems are not based on components or clearly defined interfaces, differences in the framework used can also have a negative impact on compatibility. Ideally systems are compatible to such an extent that it is possible to pick the best components from both resulting in an Instant Merge. However this situation has not been observed among the cases even though some systems share a common ancestry or are based on common standards. In reality systems may be somewhat compatible possibly allowing for an Evolutionary Merge, but there are also the options to also Choose One or Start from Scratch. If systems are totally incompatible neither Evolutionary Merge nor Instant Merge is possible. We want to emphasize that compatibility is a greyscale and there is no universal compatibility measure. Similar structures seem to be a necessary condition as well as similar in the sense of environment that defines

components [LCLB05]. What compatibility may mean in every new situation must be evaluated by the architect.

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 all systems can be retired, all integration strategies are possible. If not all systems can be retired, it is not possible to Start from Scratch because this would require discontinuing all systems. If none of the existing systems can be discontinued, both of the integration strategies Choose One and Start from Scratch are excluded. Table 4.2 summarizes the exclusion of strategies based on these concerns, where black denotes exclusion of a strategy. Although compatibility is a continuous scale, for the sake of discussion it is divided into three classes: High, Modest, and None. “High” means that the systems are compatible to such an extent that components can be picked from either one and combined into a new system with very little modification, “None” means that the systems are fundamentally different, and “Modest” is somewhere in between. Retireability is divided into All, Not all, and None. “All” means that it is possible to retire all systems, “Not all” means that out of the two or more systems one or more systems can be retired, but at least one can not, and “None” means that none of the systems can be retired.

We note that evaluating compatibility mainly involves finding information and facts about the current state of the systems, which is not the case for statements about retireability. To what extent a system is retireable is not solely determined by the architect, but involves the opinions of other stakeholders such as management, users, and marketing. This also means that retireability is not a static property, but that it can be renegotiated with involved stakeholders. This is especially true when this concern excludes a strategy that is for other reasons considered desirable. Although many influences on the decisions made were found in the cases, these two are the only ones that result in the exclusion of strategies. It appears as other influences, such as satisfaction with the existing systems, scope, available development resources, and availability of commercial products, can not be considered in isolation to exclude some strategies, but taken together they can motivate the choice of one strategy over another, and also influence retireability considerations. General influences found are discussed in section 4.5.3.

		SFS	CO	EM	IM
Compatibility	High				
	Modest				
	None				
Retireability	All				
	Not all				
	None				

Table 4.2: The exclusion of possible strategies based on concerns (black indicates exclusion)

	Retireability	Compatibility
A	All	None
B	Not all	None
C (initial)	None	Modest
C (final)	Not all	
D_{HMI}	Not all	None
*D_{Server}	(?)	Modest (?)
E1	All	Modest
E2	Not all	Modest
F1 (initial)	No	None
F1 (final)		
F2_{Pre}	All	None/Modest (?)
*F2_{2D}	All	Modest
F2_{Post}	All	None
*F2_{3D}	None	Modest
F3	All	None/Modest (?)

Table 4.3: Concerns per case (question mark indicates that the information was not available)

4.5.2 Revisiting the Cases

Table 4.3 summarizes the concerns for all cases. Table 4.4 combines tables 4.2 and 4.3 by showing which strategies are excluded, marked with black, based on the concerns, which was desired and (where applicable) eventually chosen, which is indicated with a circle. Three rows in the tables (D_{Server} , $F2_{2D}$, and $F2_{3D}$) are marked with an asterisk (“*”), indicating that the systems are not yet integrated and the information summarized is preliminary. A question mark indicates that the classification is unsure, but 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 in Table 4.4 with multiple entries for these cases showing these iterations.

4.5.3 Strategy Exclusion and Selection

In section 4.5.1 we presented two important concerns for selecting a strategy and proposed that these concerns can exclude the selection of certain strategies (e.g. if systems are not compatible they can not be instantly merged). Table 4.4 shows that most cases have selected a strategy that, according to this model, is not excluded. In these cases integration was successful or is making progress. There are three rows (C’, F1’, and F1”) where a strategy was desired that was excluded, and there have in fact been significant problems due to that: in case C the decision had to be changed, and in case F1 all alternatives are still excluded, and no significant progress has been made. Thus, two cases directly support our premise that the concerns compatibility and retireability exclude certain strategies, and the other cases do not contradict it.

	SFS	CO	EM	IM
A	O			
B		O		
C (initial)				O
C (final)		O		
D_{HMI}		O		
*D_{Server}	(?)	(?)	O	(?)
E1	O			
E2		O		
F1 (initial)			O	
F1 (final)		O		
F2_{Pre}	O		(?)	
*F2_{2D}		O		
F2_{Post}	O			
*F2_{3D}			O	
F3	O		(?)	

Table 4.4: Possible and desired strategies, per case (black indicates exclusion, circle indicates selected strategy)

The rest of this section describes observations concerning architectural compatibility, followed by a number of influences to retireability and the final choice of strategy.

Compatibility

Compatibility, unlike retireability, is not a concern that can be negotiated or modified because it is a static property of a collection of systems. Given an assessment of architectural compatibility, it cannot be changed only because it gives a dissatisfactory answer. There are two things however that can be done to improve the compatibility in order to make a merge possible. First, 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. Second, it is possible to evolve one or all systems towards a state in which they are compatible, i.e. performing an Evolutionary Merge. However, given the time required, some other strategy may be considered preferable, as shown by case C.

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 observations from the cases are provided in the following, which can be a complement to other reports of architectural incompatibility [GAJ95].

Similar high-level structures seem to be a pre-requisite for a Merge, i.e. if there are components with similar roles in the existing systems. 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. Reuse of components and architectural solutions into a common system in the cases is analysed in depth elsewhere [LCLB05].

Similarity of frameworks could also be one measure 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”. In case C, the hardware topology and communication standards define one kind of framework. One common source of incompatibility in systems is differences in the data model. Both syntactical and semantically differences can require vast changes in order to make system compatible. This has been a recurring problem in case F1. In case F2, a new data model was defined and the existing systems adapted. In case F3, the three existing systems all implemented similar workflows, however the phases were different.

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

Retireability

Retireability, unlike compatibility, can be reevaluated or renegotiated. While all cases considered the retireability of their existing systems this is an integral interwoven part of the decision process and is often not done explicitly. However it appears that cost plays an important role in the decision on retireability. Cost has many aspects; we will discuss several of them.

Existing systems represent a value to the company. Throwing away something that is known to work is often not an appealing option because it would mean discarding part of the investment. Some of the systems discussed represent hundreds of person-years of development. Also discarded parts would have to be replaced, requiring another

investment. The organization, at present, may not have the necessary resources to do this. In case F2 implementation and validation of a 3D simulator (Start from Scratch) would take the better part of 10 years, while an Evolutionary Merge, even though the systems were not totally compatible, was estimated to take less time and therefore preferred.

Another aspect that affects retireability is satisfaction. Satisfaction is very broad and can involve many stakeholders (architects, users, management, etc.) and many aspects (functionality, architecture, performance, modifiability, etc.). When one or more of the existing systems are considered dissatisfactory there is a tendency to favour replacing the dissatisfactory system(s). If some of the existing systems are considered aged, they are candidates for retirement, i.e. Choose One or Start from Scratch, as was the case for the HMIs in case D ($I_{Db:3}$).

Selection of a Strategy

Start from Scratch can be implemented as either build in-house or acquire an external system. If the domain is mature and the software systems are not core products but supporting the organization, it may be well worth to transition to a commercial or open source solution. This was the case in case F3, where a commercial software issue tracking system was acquired.

Most organizations have formalized processes for development, evolution, and retirement of software systems, and these strategies can be cast in these terms. No Integration means the existing systems are evolved independently. If Choose One is possible to implement, it appears to be the simplest strategy as it only involves retiring some systems. Start from Scratch means planning the retirement of the existing systems while starting a new development project. Evolutionary Merge cannot readily be expressed in terms of existing processes, which might indicate that it is more difficult to implement, especially since it often involves a long time scale. Once again though, the costs must be weighed against other influences. 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.

The question ‘Do we want one common system?’ is a very fundamental question and a positive answer to this is usually the starting point of the integration project. One fundamental reason we do not have an example where No Integration was selected explicitly is that the selection of cases was based on their actually trying some sort of integration. However, case F1 illustrates the situation when the existing systems are incompatible and cannot be discontinued, which leaves only strategy No Integration. In this situation, either retireability has to be reconsidered (together with the associated costs, assignment of resources, etc.) or the fundamental question about the value of integrating at all has to be revisited ($I_{F1b:3}$, $I_{F1c:9}$).

4.5.4 Feedback

There are numerous sources of influences both on whether any of the existing systems can be retired and on the selection of a strategy. It is almost impossible to distinguish between how the factors mentioned influences the retireability decision, which is often not made explicitly at all, and how these factors were re-evaluated given the resulting set of possible strategies. In case C, this feedback was fairly explicit: faced with the time needed for an Evolutionary Merge (the only available possibility), it was decided that retiring one system would cause less harm, and the decision could be changed to Choose One. However this decision took quite some time to reach resulting in a cost of at least several hundred person-years. Although is it impossible to predict what the outcome would have been if the decision to retire was taken earlier it is likely that it would have saved a lot of time and thus money. In many other cases analyses, decisions, and reconsiderations were more interleaved. We believe that all influencing factors involved in this feedback should be based on a proper analysis, documented explicitly, and made in a timely manner.

4.6 Related Work

Existing research in related areas is presented below, starting with software integration, continuing with software architecture and architectural evaluation methods, and strategic planning.

4.6.1 Software Integration

In our previous literature survey [LC04], we found that there are two classes of research on the topic of software integration:

- Basic research describing integration rather fundamentally in terms of a) interfaces [IEE90, Weg96, WK99], b) architecture [BCK03, GS93, HNS00], architectural mismatch [GAJ95], and architectural patterns [BMR⁺96, GHJV95, SSHB00], and c) information/taxonomies/data models [Gua98]. These foundations are directly useful in this context.
- There are three major fields of application: a) Component-Based Software Engineering [CL02, MO01, Szy98, WHS01b], including component technologies, b) standard interfaces and open systems [MO01, MO97], and c) Enterprise Application Integration (EAI) [Cum02, RMB00]. These existing fields address somewhat different problems than ours:
 - 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.

- These fields typically assume that components (or systems) are acquired from third parties controlling their development, meaning that modifying them is not an option. We also consider systems completely controlled in-house, and this constraint consequently does not apply.
- 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 (while not sacrificing quality).

There is no existing literature that directly addresses the context of the present research: integration or merge of software completely controlled and owned within an organization. While we certainly can and should utilize the knowledge and approaches of these fairly mature fields, our research fills an apparent gap.

The architect is considered being the person who understands the language and concerns of other stakeholders [SS02, wic02], and/or the person who monitors and decides about all changes being made to the system to ensure conceptual integrity and avoid deterioration [Par94, vGB02]. The academic focus of software architecture has been the (static) structure of the system, in terms of “components” (or “entities”) and “connectors” [BCK03, GS93, PW92]. The present paper describes many important issues an architect needs to consider during in-house integration, which only partly involves the structure of the existing systems.

There are several proposed methods for architectural analysis, mainly designed to be used during new development, such as the Architecture Trade-Off Analysis Method (ATAM) and Cost-Benefit Analysis Method (CBAM) [CBB⁺01]. Closely related to our description of compatibility is the seminal “architectural mismatch” paper, which points out many issues to be assessed as part of the architectural compatibility [GAJ95]. Also related to assessing architectural compatibility are architectural documentation good practices [CBB⁺02, HNS00, Gro00]. It has been observed that a similar structure of the existing systems is a necessary but not sufficient condition for compatibility [LCLB05]. There exist catalogues of generally useful structural patterns, however these are specifically intended for use during new development [BMR⁺96, GHJV95, SSHB00].

Many issues are not purely technical but require insight into business, and many decisions require awareness of the organization’s overall strategies. Strategic planning (and strategic management) is known from business management as a tool for this kind of reasoning, that is to systematically formulate the goals of the organization and compare with the current and forecasted environment, and take appropriate measures to be able to adapt (and possibly control) the environmental changes [Cou01, JI99]. 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 [Por98]. (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.)

4.7 Summary and Conclusions

In this paper we present strategies an organization may select when faced with two or more systems with similar functionality, of which the source code is available and changes can be made. The strategies are based on actual integration projects and are: No Integration, Start from Scratch, Choose One, and two types of Merge: Evolutionary Merge and Instant Merge.

No Integration describes the situation in which no process towards a common system is made. Start from Scratch involves either a new development process or the acquisition of a commercial product while retiring the old systems, Choose One means selecting one system to replace the others, and Merge will see components from both existing systems combined into the new system that replaces them. Evolutionary Merge seems to be the most complex strategy to implement, and Instant Merge seems to be theoretical, possible only in the rare situation of systems with very similar structures, built using the same or very similar technologies, standards and other conventions. Instant Merge strategy was the only strategy not observed in the cases, although it appears this sometimes is what management has in mind when demanding a merge of the existing systems into one.

There are two concerns to consider which will limit the set of strategies that can possibly be selected, namely architectural compatibility and retireability.

There is no exhaustive definition of architectural compatibility, but some observations in the cases are that the structures, data models, and environment that defines the components must be similar. Standards and a common ancestry can make systems somewhat compatible, although it is not a guarantee for total compatibility. Besides that compatibility is very system specific and should be thoroughly analyzed by the architects before choosing a strategy to avoid problems during integration. With somewhat compatible systems, Evolutionary Merge is possible but Instant Merge is not. If the systems are not compatible at all neither of the Merge strategies are possible.

Determining the retireability of the existing systems depends on numerous factors. Major influences found in the cases were investments made, cost, satisfaction, time to market, and available resources. Typically there is a tight feedback loop between evaluating these influences and the resulting possible strategies. Hurdles in this feedback loop, either because architects are unable to communicate their findings to management or because management delays taking decisions, cause the integration cost to increase. If it is not possible to retire all existing systems this excludes Start from Scratch as a possible strategy. If some of the existing systems can be retired, it is possible to Choose One. If none can be retired this leaves Merge as the only possibility.

The worst case scenario is that the existing systems are considered impossible to retire and are also incompatible. This leaves no strategy left but No Integration, which of course brings none of the potential benefits from integration. This was observed in one of the cases.

Drawing on the experiences from the cases, we suggest that future architects together with other stakeholders make these influences explicit thus making strategy selection

faster, better founded, and decrease the cost associated with uninformed decisions. The message to management is that delaying a decision comes with a cost.

4.7.1 Future work

We have also analysed the reuse of components and architectural solutions into a common system elsewhere [LCLB05], as well as good practices for the strategy selection process [LLC05]. We would like to compile a more complete catalogue of different aspects of architectural compatibility, for example by investigating structural patterns and styles [BMR+96, GHJV95, SSHB00] suitable for Merge, the environment defining the components, and the impact of cross-cutting concerns whose implementation is scattered throughout the system. This would also provide insight in how to make Evolutionary Merge closer to the desired Instant Merge.

4.7.2 Acknowledgements

We would like to thank all interviewees and their organizations for sharing their experiences and allowing us to publish them. We would also like to express our gratitude for the helpful suggestions from the anonymous reviewers.

Chapter 5

Validation of Research

This chapter describes the validation of the research described in the previous three chapters by means of a validating questionnaire among software engineers/architects in industry.

5.1 Introduction

Previous research has resulted in several unanswered questions. First several methods for assessing compatibility between component based on either attributes or formal models exist. Some of them include a prototype tool to assist in using the method. However whether these methods are or can be used in real-world software engineering remains open.

Second a process model for a component assessment has been proposed. It is called eXtended Compatible COTS Components Selection Method (xCCCCSM) and based on a model by Bhuta and Boehm (chapter 3) extended using the experience from two case studies. However the validity of the process model can not be established based on these two cases.

This chapter describes a questionnaire aimed at answering the two questions above and validating the research in the previous chapters by researching whether the information found is representative and therefore can be generalized to software engineering in general.

This chapter describe the goal of the research, the method used to perform the research, the design of the questionnaire, and the results. The method used is described in section 5.2. The goal of the research which led to the questionnaire is described in section 5.3. The design of the questionnaire is import to demonstrate its validity, however readers interested only in the outcome can skip its description in section 5.4 and continue with section 5.5 which contains a detailed description of the results obtained from the questionnaire. The results are summarized and discussed in the conclusion in section 5.6.

5.2 Research Method

The validation is done through a questionnaire distributed among software engineers. Since the questionnaire aims to validate the research, the questions are formulated in a neutral, non-leading way, as to not influence the respondents.

From the conclusions of the previous chapters a number of hypotheses were created. Based on the hypotheses the questionnaire was constructed. The first part of the questionnaire deals with the respondent's background and company in order to allow the results to be put in context. The other parts contains questions about different aspects of a component assessment.

The questionnaire and the raw results are included as appendix C and D in order to allow for independent verification of the conclusions.

5.3 Goal

The state-of-the-art survey in chapter 2 and the xCCCSM method in chapter 3 are founded on theoretical research, and although the xCCCSM method is based on interviews with practitioners, neither the surveyed techniques or the proposed method necessarily has a relation with real-world software engineering.

In order to establish the relationship between this research and real-world software engineering a validation of at least xCCCSM is necessary. In order to do that I first present a number of hypotheses based on the previous chapters.

Hypothesis 1 Prototyping can lead to the conclusion that the component(s) under study are unsuitable.

This hypothesis validates the addition of step 6b in the xCCCSM, which states that the results from the prototyping should be evaluated and that this can lead to none of the components being considered suitable. Rather than the original CCCSM where prototyping is assumed to always result in a suitable component.

Hypothesis 2 Published techniques to assess compatibility between components are not used.

This hypothesis states that none of the techniques found in the survey described in chapter 2 are used in software engineering in practise. Although it would be nice if this hypothesis was proven wrong, the interviews with the two engineers and two consultants in chapter 3 suggest that that is not the case and that this hypothesis is very true not just for the techniques surveyed, but for published techniques in general.

Hypothesis 3 The information provided by vendors about components does not contain the information necessary to establish compatibility.

In order to analyse of compatibility, either in a formal or informal way, certain information is required. This information includes the architecture of the component, or at the very least its design, and often also requires non-functional properties, such as performance. The hypothesis states that such information is not provided by vendors and thus architectural compatibility is difficult to assess from the information provided.

Hypothesis 4 The formality of the assessment process depends on the importance of components to the organization

The formality of the assessment process is indicated by the amount of effort spend on the process of requirements analysis, gathering information, assessing and testing components. Indications for a formal process are documented and prioritized requirements containing not only functional aspects but also non-functional once such as performance, the use of guidelines, methods, and external expertise. The opposite of a formal process is a more ad-hoc process in which requirements are refined informally, based on the available components.

The hypothesis states the higher the risks for the organization are in case the assessment results in an incorrect decision the more formal the process will be. Examples of high risk related to components are a major product of the organization depending on the success of a single component, or a product which uses a large number of components, or for which a large part of the functionality is implemented by components.

Hypothesis 5 Vendor support depends on how lucrative the deal is for the vendor

All interviewees in chapter 3 indicates that vendor support depended on how lucrative the potential deal with the assessing company would be when their component was selected. The increased vendor support was noticeable in the speed the vendor responded, the extend of the extra information provided, the extend to which the vendor participated in prototyping, and the price the vendor required for these additional services. In one interview a very lucrative deals were supported with free access to customized information, in another with assistance in prototyping free of charge.

5.4 Design

The initial idea was to validate the research by the author performing a small integration project testing the compatibility assessment techniques and assessment method. However many of the hypotheses can only be fully assessed when tested in a larger project. Especially those about the responses by vendors towards larger companies can not be verified by a small one-person project. Instead a questionnaire specifically aimed at validating the research is a better way of validating the research.

In order to be properly validate the research the questionnaire needs to be specific and answered by a large differentiated group of software engineers. To make answering the questionnaire as easy as possible it was decided to construct an digital questionnaire,

easy to fill out, and fast to return. The added bonus is that data entry is not necessary, excluding a possible source of errors and reducing time required.

Traditionally digital questionnaires have used web forms to ask questions, however this requires the participant to actively go to the website and identify himself in some way. These additional steps for a barrier, which can reduce the response rate to the questionnaire. In stead the questionnaire was designed as an Acrobat PDF file, which has the possibility to embed interactive form fields, and to return the answers by mail in a very transparent way to the user.

The full questionnaire can be found in appendix C.

5.4.1 Questions

General

The first 11 questions are designed to give a general overview of the respondent and the company he works for. They include questions about the number of years of experience, (1, 2) the respondents current position (3), the size of the company in number of developers (4), the domain(s) the company is active in (5), the percentage of projects that succeed within reasonable time and budget (6) and whether they use third-party components (7). If the answer is positive he is asked how much of the end products consists of such components (11). If they do not use third-party components the respondent is asked whether he uses internally developed components (8), why he does not use third-party components (9) and asked whether he has considered using them (10). If the answer to the latter is positive he continues with the remaining questions, if it is not he is asked to skip to the last question and submit the form.

After that, through out the questionnaire the respondent is asked how much time is spent on an activity and how successful he considers himself/the company to be at this activity. These two questions are asked for the following activities: gathering information (36, 37), component assessment (44, 45), testing (54, 58), and integration (63, 64). For integration the respondent is also asked to judge how successful projects are on average (62).

The respondents have the possibility to add comments in specifically designed comment boxes at the end of every section. At the end of the questionnaire the respondent is specifically asked to add any additional information or comments (65).

Hypothesis 1

Prototyping can lead to the conclusion that the component(s) under study are unsuitable.

The main extension of the original CCCSM in xCCCSM is the addition of the evaluation step after the prototyping stage. This step is necessary because in the cases prototyping let to more information then was uncovered while gathering information and let to the conclusion that the components were not suitable. This hypothesis aims to support

that assumption. Questions 56 and 57 are designed to confirm the hypothesis by asking whether testing in general, including prototyping, has let to the a promising component being considered unsuitable. The questions leading up to 57 (specifically 46-50, 55) are designed to gather specifics about the kind of prototyping and testing that is performed. Finally questions 59 and 60 inquire whether problems are encountered during integration and whether they could have been prevented with more testing and/or prototyping.

Hypothesis 2

Published techniques to assess compatibility between components are not used.

In a general questionnaire aimed at a diverse population it is difficult to ask whether developers use formal methods, since the definition of what constitutes a formal method might not be known, or uniformly understood. Therefore questions 39 and 40 ask whether the engineer uses methods and/or techniques to assist in the assessment and if so which ones to provide the widest possible range of answers, which will hopefully be understood to include formal methods if they are used in the company. The risk with asking such an open question is that respondents do not know what to answer or skip to question more easily than they would a closed question. However considering the circumstances no more specific question can be included.

Related to this hypothesis is question 38, which asks whether respondent use guidelines for performing the assessment, which might indicate that the company mandates some kind of process, technique, or tool.

Hypothesis 3

The information provided by vendors about components does not contain the information necessary to establish compatibility.

In order to determine whether the information provided can actually be used for architectural compatibility analysis several questions are asked intended in obtaining the type, source, and quality of the information that comes with components. Question 26 asks where the initial knowledge of the existence of the component is obtained. Question 27 asks about the type of information provided with the component, suggestion several alternatives and providing a large input field for additional information. Questions 28, 29, and 30 ask for the quality, completeness, and correctness of the documentation provided. Finally questions 35 inquires to additional sources of information.

Hypothesis 4

The formality of the assessment process depends on the importance of components to the organization.

The questionnaire contains 19 questions about the process followed while assessing components.

Starting off the respondent is asked how much of the end product consists of third-party components (11). Concerning requirements the respondent is asked whether he compiles an extensive requirements specification before searching for components (12). The next question they need to answer depends on whether they do or not, but the questions are essentially the same, but asked from a different perspective. Questions include whether requirements include non-functional requirements (14, 21) and whether requirements are changed during development (16, 17, 22). If a detailed specification is created the respondent is asked whether requirements are prioritized (14) and whether a less thorough process would suite better (18). If no detailed requirements are specified, but some basic requirements exist (19) the respondent is asked whether they are written down (20) and whether requirements are evolved while selecting components (22, 23). Also the respondent is asked why no detailed requirements specification is created (24) and whether he feels writing a requirements specification would improve the process (25). The latter two are also asked when no requirements exist.

The four questions about the process of assessing components include whether guidelines exist (38), which methods and/or tools are used (39, 40), if the processes is iterative (41) and whether non-software related expertise is involved (42, 43).

Finally question 61 asks whether the requirements and/or scope of the project change based on problems encountered during integration.

Hypothesis 5

Vendor support depends on how lucrative the deal is for the vendor

Determining whether an organization is lucrative for a component vendor is difficult to determine in a general survey like this one. However we assume that this can be deferred from the company size (4) and the amount of components used (11).

In order to determine whether the company size has influence on the vendor the questionnaire contains 7 questions, 4 about gathering information and 3 about prototyping.

We would like to know whether more information then provided is necessary to make the initial assessment (31), if so how often is this the case (32), how willing is the vendor to provide this additional information (33), and how fast is it provided (34).

Regarding prototyping the questionnaire inquires as to whether vendors assist in prototyping (51), and if so whether this service is provided free of charge (52), and whether this results in additional information (53).

5.4.2 Distribution

The questionnaire is distributed to two distinct groups of people. The first group are personal contacts of the author including architects, researchers and developers at companies developing embedded system for industrial use or consumer electronics, and consultants either working at a consulting firm or independently. In total this group consists of 19

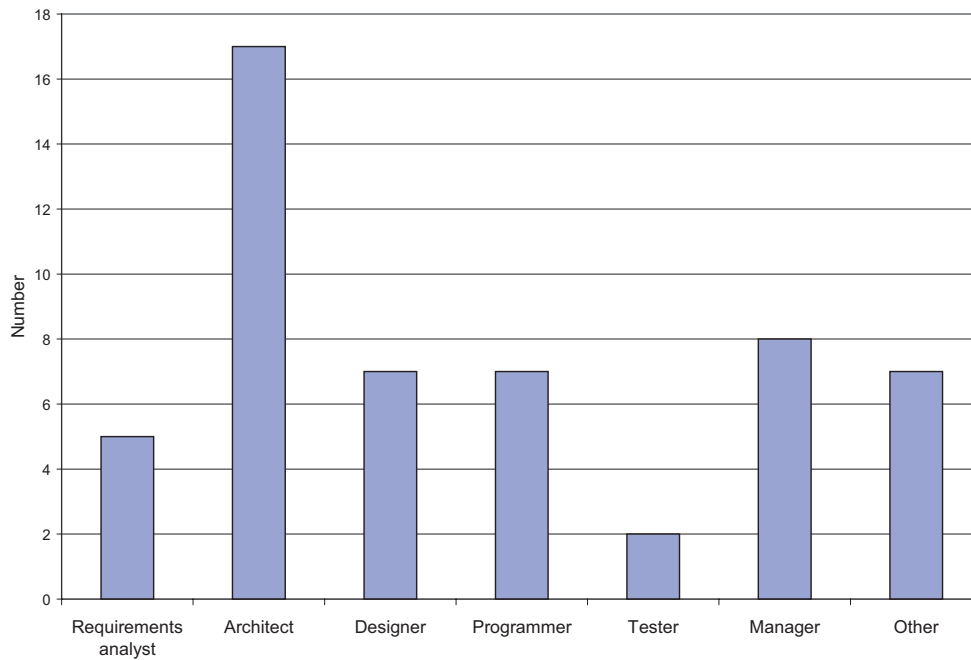


Figure 5.1: Current position

people, several persons in key positions were also asked to distribute the questionnaire to people they thought would be interested in filling it out.

The second group consists of companies in the IT sector of Teknikbyn, Västerås Technology Park. These are in total 23 companies and the questionnaire was sent to email addresses gathered from the respective websites, in total 38 emails.

5.5 Results

5.5.1 General

Questions 1 through 6 were designed to obtain information about the respondents and his company. A large majority of the respondents (74%) has 10 years of more in software development. The other six have between 1 and 10 years of experience, none of the respondents have less than 1 year of experience. When looking at the number of years a respondent has been in his current position this is fairly evenly distributed over the categories up to 10 years. Only 2 respondents have been in their current position for over 10 years, however both work as consultants in fairly different organizations.

Figure 5.1 shows the respondent's positions. The total number of responses is higher than the number of respondents because multiple answers to this question were possible. The results show that most respondents are working as architects. This is most likely due to the fact that most questionnaire were distributed to people which are known software architects as described in section 5.4.2. Although the respondents clearly do

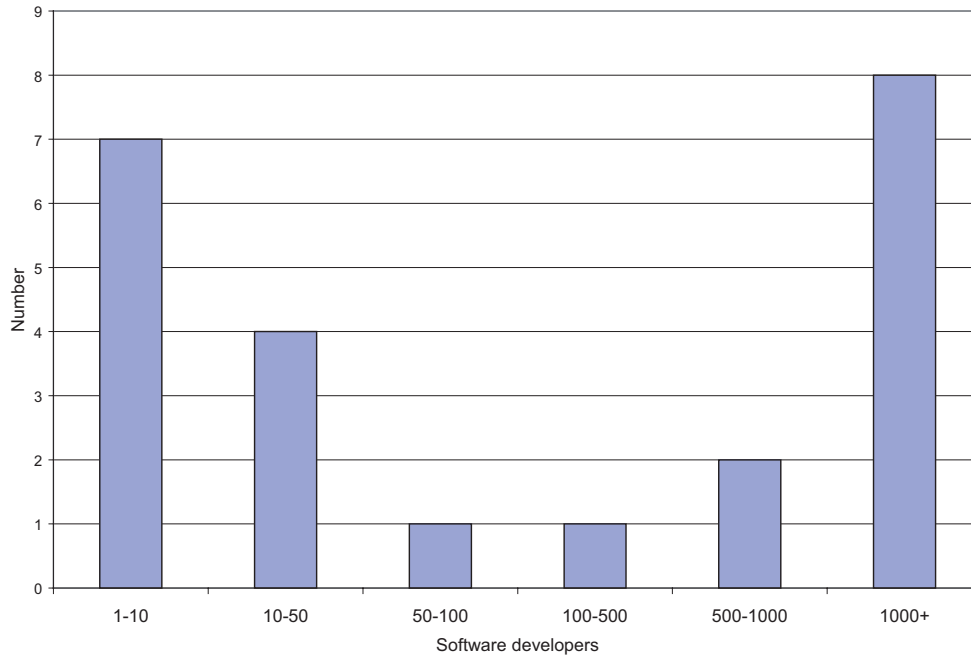


Figure 5.2: Company size in number of software developers

not represent a cross-section of software engineers the fact that they are familiar with software architecture and the fact that most of the respondents have also occupied other jobs in the past should allow for generalization of the results. The most common jobs combined with architect are manager (7 times), designer (6), programmer (6), and requirements analyst (5). Under “Other” the following professions are listed: researcher (4 times), security specialist (2), and process improvement change agent (1).

The company size in number of software developers is plotted in figure 5.2, showing that a large majority of respondents (65%) are either working in very small or very large companies. This is again caused by the means of distributing the questionnaire. Because only very few medium sized companies are reflected in the responses to the questions this potentially makes it difficult to generalize all results to companies of all sizes. However in some cases we argue that a result that holds for both small and large companies also holds for medium sized companies. The size of the company the respondent works for is not always representative for the company he performs the assessments in. At least one respondent works for a one-man consulting company but exclusively consults for very large organizations. The other way around, in which architects working for larger companies are mainly involved in projects for smaller companies, might also be possible, but there is no known occurrence of that among the respondents.

As shown by graph 5.3 the respondents work in all domains of software engineering. Almost all respondents indicated that they worked in multiple domains. Two indicated that they work in all domains. Common combinations are, not surprisingly, real time and embedded (7 times), and business systems and consulting (5 times). The combination business systems and either real-time or embedded occurs 3 times. Only one respondent

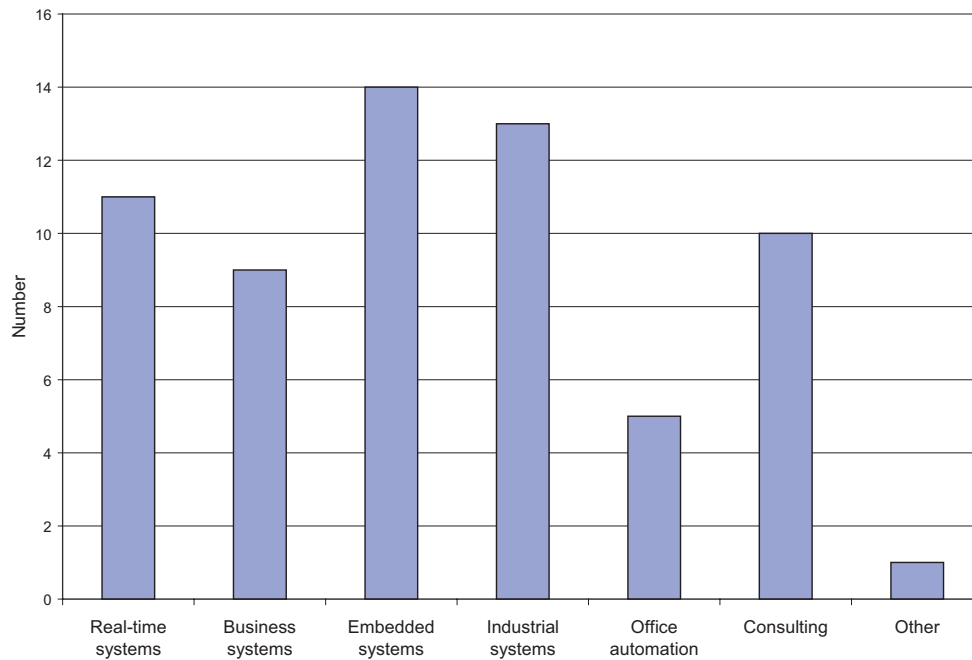


Figure 5.3: Domain

did not find any of the domains matching his field, which is developing software for simulations to support quality of the core product of his company.

Question 6 was designed to determine the maturity of the organization by asking how many projects were completed within set time and budget. Since only very few projects actually are, the questions was formulated to ask how many projects where completed within twice the time and twice the budget. About 1/3 is unknown, in over 1/3 of the cases 75 to 100% of the projects are completed within the specified time and the other third less than 50% of the projects are completed within the specified time and budget.

Of the 23 two respondents filled out that they had Never used or considered using components, the other 21 did use components. The remainder of this section will deal with the answers from those respondents that did use components.

Figure 5.4 shows on average how much of the end product consists of third-party components. The answers vary greatly, and several respondents note in a comment that it is difficult to say because it varies much between projects. It is worth noting that the respondents that implement an end product with more than 50% third-party components all work for very large organizations (≥ 500 developers) that have recognized the importance of components and have developed specific expertise in this area. The respondents that use between 0 and 10 percent third-party components all work for small (< 50 developers) companies. However this can not be generalized because the survey also shows small companies using between 25 and 50% components and large companies using between 10 and 25% components.

The pie chart in figure 5.5 shows how many respondents write requirements before search-

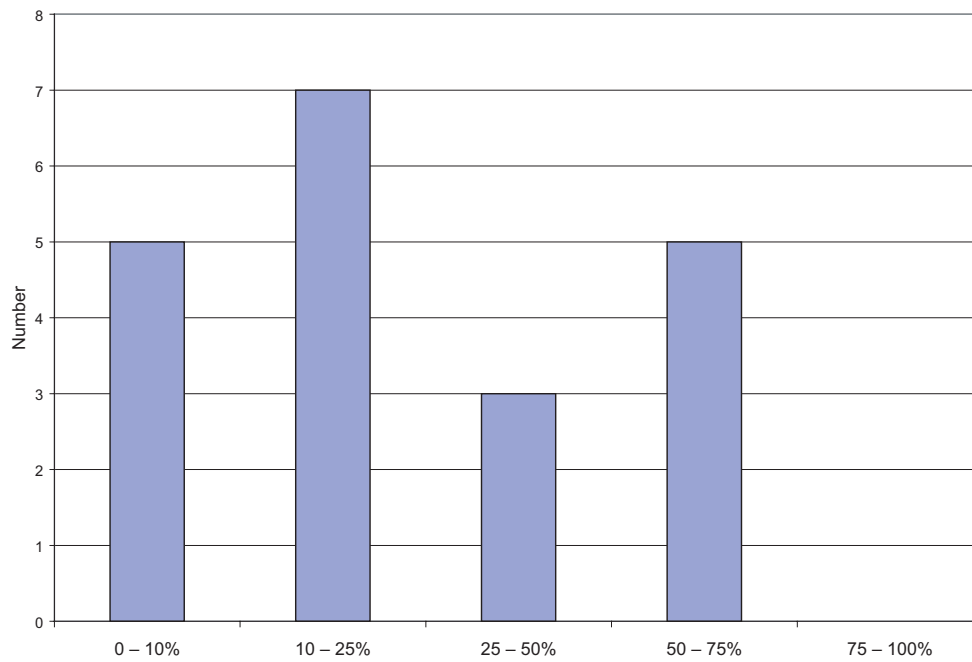


Figure 5.4: COTS usage

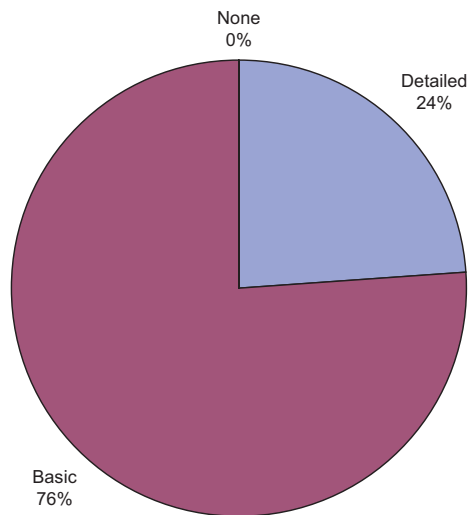


Figure 5.5: Requirements

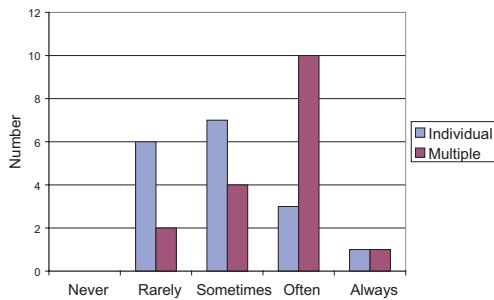


Figure 5.6: Prototyping individual vs multiple components

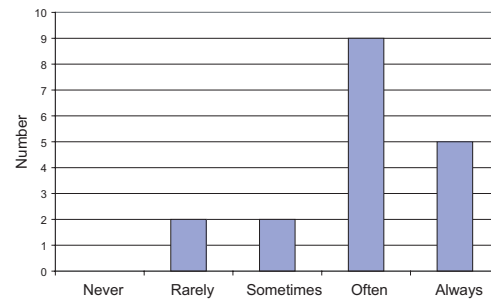


Figure 5.7: Prototyping changes opinion

ing for components and what kind of requirements. About one fourth write detailed requirements and three fourths basic requirements. These basic requirements are more than Often documented. None of the respondents starts looking for components without at least a basic idea of the requirements.

5.5.2 Testing the Hypotheses

Hypothesis 1

Prototyping can lead to the conclusion that the component(s) under study are unsuitable.

Out of the 21 respondents that use third-party components, all but two test these components. Of the remaining 19 all but one uses prototyping to do so. Three fourth uses prototyping Often or Always, only 3 use is Sometimes. Respondents tend to favour prototyping of combination of components over prototyping individual components, about 44% scores individual as rarely while scoring combined as Often. The median response for individual prototyping is Sometimes (3), and Often (4) for the combination, see figure 5.6.

When asked whether testing, including prototyping, changes the respondents opinion of the suitability of the components the majority of respondents answer Often or Always (74%), none of the respondents indicate Never as shown in figure 5.7. Over 50% scores Sometimes when asked whether testing leads to the conclusion that the component does not meet the requirements. Only one person scored Never, which could be explained by the small size of the components the respondent use and the fact that they tend to not interact with each other.

The fact that the questionnaire shows all respondents have had their opinion about the suitability of the component changes because of testing and the fact that the overwhelming majority (95%) of the respondents that do test use prototyping, strongly supports the hypothesis.

The results would most likely be even higher if testing was used more often or more extensively, as indicated by the responses to question 60: over 80% of the respondents

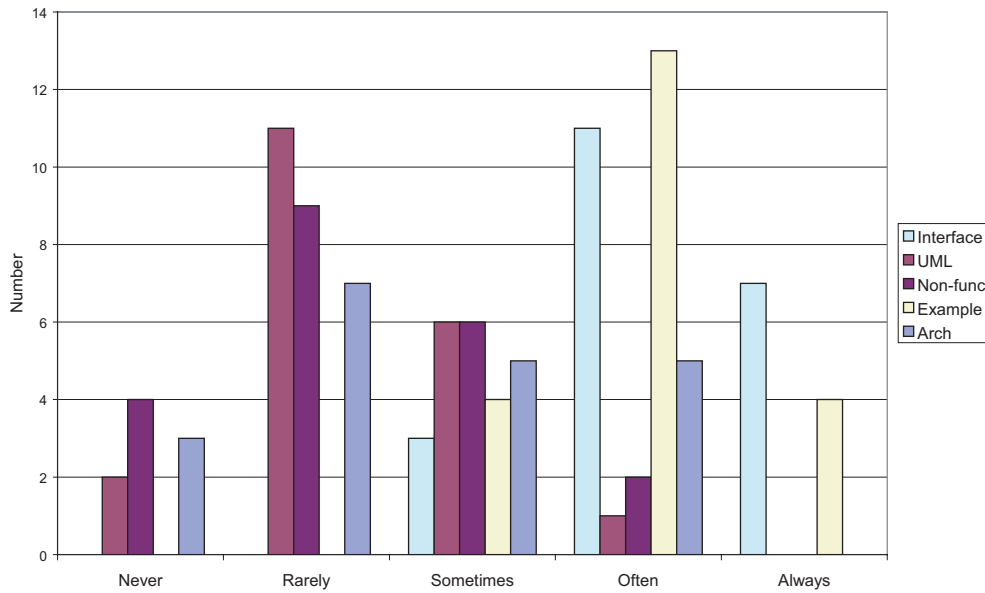


Figure 5.8: Available documentation

indicate that this could prevent problems during integration. Something all experience sometimes, and a large minority (38%) Often or Always.

The results clearly support the hypothesis and thus we can state the prototyping does lead to new insights and can lead to the conclusion that the component is unsuitable after all.

Hypothesis 2

Published techniques to assess compatibility between components are not used.

Eleven (52%) answer that they never use methods and/or tools to assist in component assessment. Three (14%) use them Often, in all three cases the methods used are company mandated checklists or guidelines. All other (33%) use them Rarely or Sometimes.

So although one of the respondents mentioned a scientifically based technique for assessing economical tradeoffs in components none mentioned any technique related to assessing compatibility. Therefore we can conclude that the hypothesis is strongly supported and no techniques such as the ones described in chapter 2 are used among the practitioners that were surveyed.

Hypothesis 3

The information provided by vendors about components does not contain the information necessary to establish compatibility.

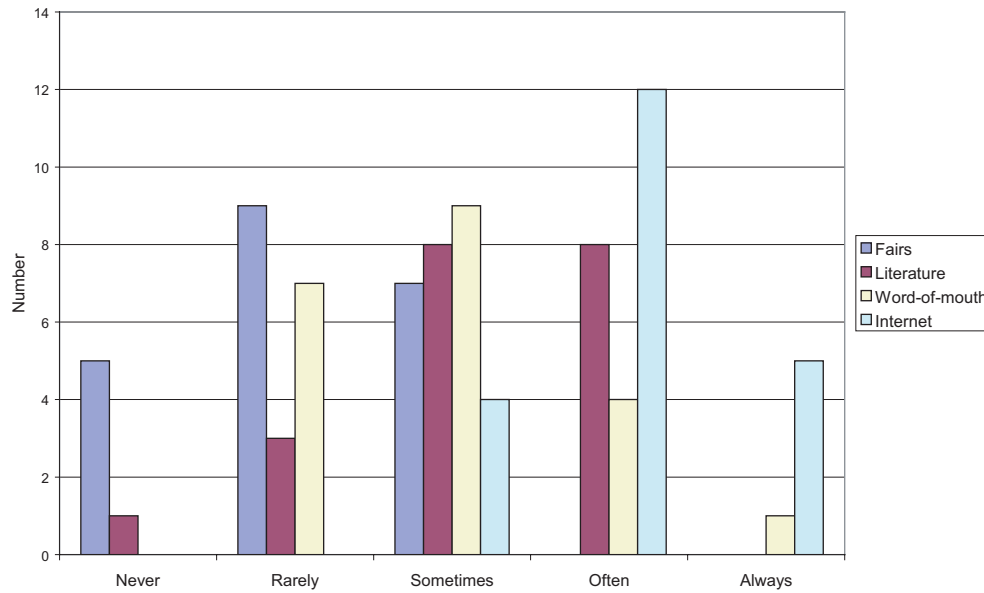


Figure 5.9: Initial information

Question 27 was specifically designed to answer this question by asking which kinds of information are provided with a component and allowing the respondents to score this 1 (Never) to 5 (Always). The differences for the types of documentation are significant as shown in figure 5.8. For both interface description and usage examples over 80% scores either Often (4) or Always (5), the remaining respondents score Sometimes (3). For architectural description, design documents, and non-functional documentation over half the respondents score Rarely (2) or Sometimes (3). For design documents this number is even over 80%.

The questionnaire also contained a number of questions about the quality of the documentation provided and other sources of information besides documentation. The quality of the documentation could be scored bad (1) to excellent (5).

Two-thirds of the respondents indicate that the quality of the provided documentation is average (3), only 3 respondents (14%) indicate the quality as above average, notable is that two out of these three also scored Often on the availability of architectural documentation. The scores on the completeness of the documentation are even lower, here all respondents score either below average (2) or average (3), with only slightly more than half (61%) scoring average. The correctness of the documentation is considered to be better, here 85% scores either average (3) or above average (4).

When asked where initial information about components is obtained Internet scores very high with an Often or Always in all but 4 cases (81%), professional literature scores quite high as well with half the respondents consulting it Sometimes or Often. Slightly less used it word-of-mouth which scores an average of Sometimes. Trade fairs score the lowest with only most people using the Rarely and only four Often. The results are plotted in figure 5.9.

Once a component is selected for assessment the sources of information consulted other than documentation include information from colleagues which is used Sometimes or Often in over half the cases. References are used less often, but still half the respondents use them Sometimes or more. Commercial evaluators are used the least with an average of Rarely.

The results of the questionnaire support the hypothesis that high level documentation such as architectural or design descriptions and non-functional properties are only seldom available. Since these contain the information necessary to establish architectural compatibility beforehand the hypothesis that the information necessary to establish compatibility is not provided by component vendors. Instead practitioners seem to rely heavily on experience from colleagues, either from within the organization, or as mentioned explicitly several times through the internet.

However there are five notable exceptions, that do indicate that architectural description are Often available. What three out of these five have in common is that they work for very large companies (500+ developers) and all of them indicate that the end product consists of between 50 to 75 percent third-party components.

Hypothesis 4

The formality of the assessment process depends on the importance of components to the organization.

The measure of formality used in the analysis is a summation of the scores of the questions that indicate formality minus the scores of the questions that indicate the opposite. Questions that are answered Yes/No as assigned a score between 1 and 5 depending on their impact to formality. The questions that contribute positively to the summation are (with in between brackets the assigned score for Yes/No questions): 12 (5), 13, 14, 19 (2), 20, 21, 38 (5), 39, and 42. Questions contributing negatively are: 16, 22, 41, and 61.

The formality score is set against a number of questions that are thought to have a relation with the importance of the components to the organization. These questions are:

- Company size (4)
- Percentage of third-part components in end product (11)

The assumption underlying the choice of these two variables is that larger companies have larger products, tougher quality requirements, and more demanding customers and therefore finding correct components is of more importance to them. Similarly organizations that use a large number of components rely heavier on the correct assessment of these components than companies that use fewer component. In both cases in order to reduce risk, it is expected that the organization uses a more formal approach.

In order to detect whether the formality depends on other aspects then mentioned in the hypothesis it is also checked against:

- Experience of respondent (1)
- Existence of architectural information (27)
- Vendor support (33, 34, 52)
- How good respondents consider themselves at integration (45)

The formality of the requirements gathering process and the assessment process itself were also considered separately.

From the above variables the only one that shows a positive relation with the formality of the process with a correlation coefficients of 0.5 or higher is the relationship between formality and percentage of third-party components used, which has a correlation coefficients of 0.81. None of the other variables including company size show a distinguishable correlation with the formality.

The hypothesis is supported, providing the assumptions that percentage of third-party components is a good indicator for the importance of correct components to the organization.

Hypothesis 5

Vendor support depends on how lucrative the deal is for the vendor

The questionnaire deals with vendor support when it comes to providing more assistance than normal when it comes to providing additional information (questions 31, 32, 33 and 34) and support during prototyping (questions 51, 52 and 53). As stated before the lucratively of the deal between a company and vendor is difficult to assess in a general survey like this one. However we assume that company size and percentage of components used are indicators.

Out of the 21 respondents 19 required more information from the vendor than initially provided. Of these respondents 74% needs additional information Often or Always. And only 11% needed it Rarely. The willingness of the vendor to provide additional information is considered by 79% to be average (3) or above average (4). Only 16% judges this as below average. The speed with which the information provided is not considered this high, in stead 84% considers this either below average (2) or average (3). When set against the company size or the percentage of components used there is no clear correlation between the indicating for the lucratively and the indicator for support of the vendor which is a summation of the willingness and the speed.

When looking at vendor support during prototyping the answers vary considerably between respondents but also no correlation can be found between this and either the company size or the percentage of components used.

The fact that the data does not support the hypothesis could be caused by several factors. First of all the company size and the percentage of components used might not be a good indicator for the relationship between the company and the vendor. A better

measure would be the relative size of the assessing organization compared to that of the component vendor and the amount of money involved compared to the vendor's turnover. However such data can not be obtained in a general questionnaire, only through specific case studies such as the interviews described in chapter 3.

Another aspect is that the scale used in the questionnaire is rather subjective, it could very well be that developers at smaller companies have lesser expectations from component vendors and therefore score the willingness higher.

5.6 Conclusion

The results from the questionnaire confirm the first four out of five hypotheses. Indicating that the research that the questionnaire was aimed to validate is indeed valid, i.e. the process as described is actually useful, or already partly being used, in practise.

The results once again underline the importance of prototyping, especially prototyping multiple components to learn more about their interactions. All respondents indicate that prototyping has led them to change their opinion about the suitability of a component. So although the information gathered about the component did indicate compatibility, the results from the prototyping indicate the opposite.

As assumed in chapter 2 scientific techniques for assessing compatibility of components are not used. This is most likely due to the fact the the information necessary to assess the compatibility, which includes architectural description, design document, and non-functional properties, is not available as indicated by over half the respondents scoring either Rarely or Sometimes when asked whether this kind of information was available.

The formality of the assessment process can be linked to the importance of third-party components in the organization, however the importance does not depend on the size of the company, rather on the percentage of third-party components used in the end product.

The fifth and last hypothesis (*Vendor support depends on how lucrative the deal is for the vendor*) is not supported by the response to the questionnaire. However we believe that this is due to inability of the questionnaire to ascertain the lucratively of the vendor organization relationship, rather than because the hypothesis is untrue. However still the hypothesis cannot be confirmed based on the available data, therefore more research is necessary to establish its validity.

Chapter 6

Conclusion

As stated in the introduction on page 3 the goal of this thesis is to:

“[...] unify state-of-the-art research into both techniques and processes with current established practises in industry.”

This goal has been achieved through the Extended Compatible COTS Components Selection Method (xCCCSM), a process model that describes the process of assessing (sets of) components. The model is based on work by Bhuta and Boehm and extended in order to fit current practise better based on the experience of two software engineers. Furthermore the model has been validated through a questionnaire answered by over 20 software engineers.

The model proposed a number of steps of which the most important ones are gathering information about the component(s) and prototyping the (set of) component(s). Both steps are iterative in nature. After each iteration the results are evaluated based on the requirements. Eventually this will result in either zero, or one or more sets of components being considered compatible with the requirements, each other, and the system. In case no suitable component can be found the organization will have to decide to either change the requirements or develop a component in-house. This model is depicted in figure 6.1.

Ideally one would like to assess compatibility as soon as information about the components is available. Two sets of techniques exist to facilitate this. The first group uses attributes from the different components to determine possible mismatches. This technique can handle incomplete information, however this is done in a very pessimistic way causing numerous false positives when presented with many unknowns. This is a major issue because, as the questionnaire has shown, the information necessary to determine these attributes, e.g. architectural description, design documentation, and non-functional properties, are very often not available. The second group of techniques attempts to find incompatibilities by modelling the component using a formal modelling approach depending on the architecture type of the system. The downside of this method is that it requires even more information than the attribute based techniques, for which not enough information is available as is, it requires complete modelling of the system

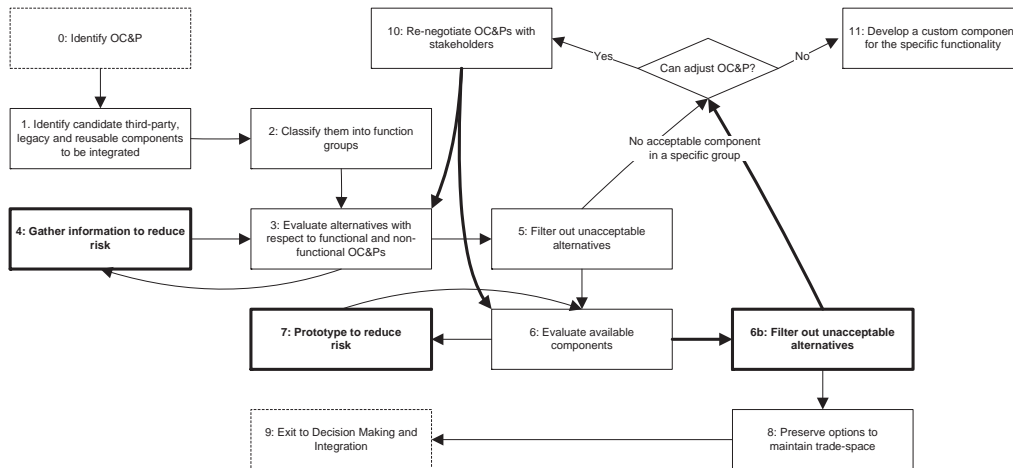


Figure 6.1: Extended Compatible COTS Components Selection Method (xCCCSM)

and all components, has a high learning curve, very limited tool support and is therefore very time consuming. These disadvantages mean that in practise these techniques will only be used in domains that traditionally spend a large effort on formal methods, domains that require very high safety or availability.

Because of the limited possibilities to evaluate components early on prototyping is often the most important step to exclude components that are incompatible. As indicated by the very high number of respondents to the questionnaire that use prototyping (95%) and the fact that all encounter situation where prototyping results in new information about the suitability of the component.

Architectural compatibility is not only an issue when assessing third-party components. In-house developed components can also benefit from compatibility assessment when several components that need to be integrated are developed in independent development organizations within the same company. An intermediate form is when companies out-source the development of parts of the software to other organizations to be developed as components. In that case the requesting organization can dictate the architectural constraints, however it will need to be fully aware of the attributes of its own components in able to do so.

Overall this thesis presented a method for assessing compatibility which combines the available theory with common practise in a way that can be used in practical software engineering.

6.1 Future Work

Although the xCCCSM method was constructed based on experience from software engineers and validated through the questionnaire, the only way to test its effectiveness it to collect more information about about what is done in each step today and to implement the method in a real-world situation under such conditions that the results can be reliably measured and evaluated.

Although the thesis shows that the techniques for assessing compatibility as early as the information gathering stage are not viable because they are still in an prototype stage and the information necessary is not available. In order to create mature techniques more research is required as is more experience in implementing them. The lack of information this can resolved be improving the handling of unknowns and the extraction of attributes from the documentation that is available could greatly benefit the detection of mismatches early on and thus the efficiency of the assessment. Alternatively component certification should require vendors to provide the necessary information. Such an approach is attempted by Predictable Assembly from Certifiable Components [[pac](#)].

Bibliography

- [AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering Methodology*, (6):213–249, 1997.
- [BB01] Victor R. Basili and Barry Boehm. Cots-based systems top 10 list. *IEEE Software*, pages 91–93, May 2001.
- [BB05] J. Bhuta and Barry Boehm. A method for compatible cots component selection. In *Proceedings of International Conference on COTS-Based Software Systems*, pages 132–143, 2005.
- [BCD00] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. On the formalization of architectural types with process algebras. In *ACM SIGSOFT Software Engineering Notes*, pages 140–148, 2000.
- [BCD01] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Detecting architectural mismatches in process algebraic descriptions of software systems. In *Proceedings of Working IEEE/IFIP Conference on Software Architecture*, pages 77–86. IEEE, 2001.
- [BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, second edition, 2003.
- [BMR⁺96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons, 1996.
- [BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering Methodology*, (1):355–398, 1992.
- [BOS00] Lisa Brownsword, Tricia Oberndorf, and Carol A. Sledge. Developing new processes for cots-based systems. *IEEE Software*, pages 48–55, July/August 2000.
- [CBB⁺01] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Evaluating Software Architectures*. Addison-Wesley, 2001.
- [CBB⁺02] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

- [CC05] Ivica Crnkovic and Michel Chaudron. Component-based development process and component lifecycle. In *Proceedings of 27th International Conference Information Technology Interfaces (ITI)*. IEEE, 2005.
- [cit] CiteSeer.IST. <http://citeseer.ist.psu.edu/>.
- [CL02] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.
- [col] Collidescope. <http://www.seat.utulsa.edu/collidescope.php>.
- [Cou01] H. Courtney. *20—20 Foresight : Crafting Strategy in an Uncertain World*. Harvard Business School Press, 2001.
- [Cum02] F. A. Cummins. *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*. John Wiley & Sons, 2002.
- [DeL99a] Robert DeLine. Avoiding packaging mismatch with flexible packaging. *International Conference on Software Engineering*, pages 97–106, 1999.
- [DeL99b] Robert DeLine. A catalog of techniques for resolving packaging mismatch. pages 44–53, 1999.
- [DFGK03] L. Davis, D. Flagg, R. Gamble, and C. Karatas. Classifying interoperability conflicts. In *Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 62–71. Springer-Verlag, February 2003.
- [DG02] L. Davis and Rose Gamble. Identifying evolvability for integration. In *IC-CBSS 2002*, pages 65–75, 2002.
- [DGP⁺01] L. Davis, R. Gamble, J. Payton, G. Jónsdóttir, and D. Underwood. A notation for problematic architecture interactions. *ACM SIGSOFT Software Engineering Notes*, 26(5), 2001.
- [EG99] Alexander Egyed and Cristina Gacek. Automatically detecting mismatches during component-based and model-based development. In *14th IEEE International Conference on Automated Software Engineering*, pages 191–199, 1999.
- [EMG00] A. Egyed, N. Medvidovic, and C. Gacek. Component-based perspective on software mismatch detection and resolution. *IEE Proceedings - Software*, 147(6), December 2000.
- [GAJ95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
- [GAO88] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT foundations of software engineering*, 175-188.
- [GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, April 1995.

- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [goo] Google scholar. <http://scholar.google.com>.
- [Gro00] IEEE Architecture Working Group. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000*. IEEE, 2000.
- [GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.
- [Gua98] N. Guarino. *Formal Ontology in Information Systems*. IOS Press, 1998.
- [HC01] G. T. Heineman and W. T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.
- [HCS⁺05] G. T. Heineman, Ivica Crnkovic, H. Schmidt, J. Stafford, C. Szyperski, and K. C. Wallnau. *Component-Based Software Engineering: 8th International Symposium, CBSE 2005*. Lecture Notes in Computer Science Vol. 3489. Springer, 2005.
- [HNS00] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
- [IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990*. IEEE, 1990.
- [iso] Technical report.
- [IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.
- [JI99] A. A. Thompson Jr. and A. J. Strickland III. *Strategic Management : Concepts and Cases*. Irwin/McGraw-Hill, eleventh edition, 1999.
- [LC04] Rikard Land and Ivica Crnkovic. Existing approaches to software integration - and a challenge for the future. In *Proceedings of Software Engineering Research and Practice in Sweden*. Linkping University, 2004.
- [LCLB05] Rikard Land, Ivica Crnkovic, Stig Larsson, and Laurens 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*. Springer, 2005.
- [LLC] Rikard Land, Stig Larsson, and Ivica Crnkovic. Technical report.

- [LLC05] Rikard Land, Stig Larsson, and Ivica 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, Software Process and Product Improvement track*, 2005.
- [LV95] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, (21):717–734, 1995.
- [Max92] Joseph A. Maxwell. Understanding and validity in qualitative research. *Harvard Educational Review*, 62(3):279–300, 1992.
- [MBF99] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad. Framework integration: Problems, causes, solutions. *Communications of the ACM*, 42(10), October 1999.
- [MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on the foundations of Software Engineering.*, pages 3–14, October 1996.
- [MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. pages 178–187, 2000.
- [MO97] C. Meyers and Tricia Oberndorf. *Open Systems: The Promises and the Pitfalls*. Addison-Wesley, 1997.
- [MO01] C. Meyers and P. Oberndorf. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley, 2001.
- [MRT99] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *Proc. 21st Intl Conf. Software Eng. (ICSE 99)*, pages 44–53, May 1999.
- [NM99] C. Ncube and N. A. Maiden. Pore: Procurement-oriented requirements engineering method for the component-based systems engineering development paradigm. In *Proceedings of ICSE 1999*, 1999.
- [NM01] C. Ncube and N. A. Maiden. *Selecting the Right COTS Software: Why Requirements Are Important*. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.
- [pac] Predictable Assembly from Certifiable Components. <http://www.sei.cmu.edu/pacc/>.
- [Par94] D. L. Parnas. Software aging. In *Proceedings of The 16th International Conference on Software Engineering*, pages 279–287. IEEE Press, 1994.
- [Por98] M. E. Porter. *Competitive Strategy: Techniques for Analyzing Industries and Competitors*. Free Press, 1998.
- [Pou96] Jeffrey S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, Reading, Massachusetts, 1996.

- [PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
- [RMB00] W. A. Ruh, F. X. Maginnis, and W. J. Brown. *Enterprise Application Integration*. John Wiley & Sons, 2000.
- [Rob02] C. Robson. *Real World Research*. Blackwell Publishers, second edition, 2002.
- [Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
- [SC98] A. Strauss and J. M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, second edition, 1998.
- [Sea99] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July/August 1999.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.
- [Sha93] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In David Alex Lamb, editor, *ICSE Workshop on Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1993.
- [Sha95] Mary Shaw. Architectural issues in software reuse: Its not just the functionality, its the packaging. In *Proceedings of the Symposium on Software Reusability*, pages 3–6, 1995.
- [Sha96] Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 181–185, 1996.
- [Som04] Ian Sommerville. *Software Engineering*. Addison-Wesley, seventh edition, 2004.
- [SS02] M. T. Sewell and L. M. Sewell. *The Software Architect's Profession - An Introduction*. Prentice Hall PTR, 2002.
- [SSC04] R. Shukla, P. Strooper, and D. Carrington. A framework for reliability assessment of software components. In *In Proceedings of Component-Based Software Engineering: 7th International Symposium, CBSE 2004, Lecture Notes in Computer Science 3054*, pages 272–279. Springer, 2004.
- [SSHB00] D. Schmidt, M. Stal, H. Rohnert H., and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.
- [Szy98] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

- [Szy02] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
- [TMA⁺96] R.N. Taylor, N. Medvidovic, K.N. Anderson, E.J.Jr. Whitehead, J.E. Robbins, K.A.Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.
- [TS06] Tim Trew and G. Soepenber. Identifying technical risks in third-party software for embedded products. In *Proceedings of the 6th International Conference on COTS-based Software Systems (ICCBSS 2006)*, 2006.
- [uml] UML 2.0. <http://www.uml.org/>.
- [vGB02] J. van Gurp and J. Bosch. Design erosion: Problems & causes. *Journal of Systems & Software*, 61(2):105–119, 2002.
- [Weg96] P. Wegner. Interoperability. *ACM Computing Surveys*, 28(1), 1996.
- [WHS01a] K. C. Wallnau, S. A. Hissam, and R. C. Seacord. *Building Systems from Commercial Components*. Addison-Wesley, 2001.
- [WHS01b] K. C. Wallnau, S. A. Hissam, and R. C. Seacord. *Building Systems from Commercial Components*. Addison-Wesley, 2001.
- [wic02] WWISA, Worldwide Institute of Software Architects. <http://www.wwisa.org>, 2002.
- [WK99] J. C. Wileden and A. Kaplan. Software interoperability: Principles and practice. In *Proceedings of 21st International Conference on Software Engineering*, pages 675–676. ACM, 1999.
- [YBB99] Daniil Yakimovich, James M. Bieman, and Victor R. Basili. Software architecture classification for estimating the cost of cots integration. In *Proceedings of the 21st International Conference on Software Engineering*, pages 296–302. ACM, 1999.
- [Yin03] R. K. Yin. *Case Study Research : Design and Methods*. Sage Publications, third edition, 2003.
- [YTB99] Daniil Yakimovich, Guilherme H. Travassos, and Victor R. Basili. A classification of software components incompatibilities for cots integration. 1999.

Appendix A

WICSA paper

Architectural Concerns When Selecting an In-House Integration Strategy – Experiences from Industry

Rikard Land^{*}, Laurens Blankers[#], Stig Larsson^{*}, Ivica Crnkovic^{*}

^{*}Mälardalen University, Department of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
{rikard.land, stig.larsson, ivica.crnkovic}@mdh.se

[#]Eindhoven University of Technology, Department of Mathematics and Computing Science
PO Box 513, 5600 MB Eindhoven, Netherlands
l.blankers@student.tue.nl

1. Introduction

Consider the scenario where two or more software systems have been developed in-house, for different purposes. Over time, the systems have been evolved to contain more functionality, until a point where there is some overlap in functionality and purpose. The same situation occurs, only more drastically, as a result of company acquisitions and mergers. A new system combining the functionality of the existing systems would improve the situation both from an economical and maintenance point of view, and from the point of view of users, marketing and customers.

To investigate this problem of *in-house integration*, we carried out a multiple case study, consisting of nine integration projects in six organizations from different domains (here labelled *A-F*). For details on methodology, a presentation of the data sources, and the complete copied out interviews, see [3]. Elsewhere we have analyzed the case study data from a process point of view [4], and discussed the possibilities for reuse in this context [2]; the present paper investigates issues of importance to an industrial architect [5] and focuses on how to select a high-level integration strategy.

2. Selecting a Strategy

The following in-house integration strategies have been identified:

No Integration (NI) Do nothing – this requires no extra effort or resources in the short term, but can consequently not give any return on investment.

Start from Scratch (SFS) Discontinue all existing systems and initiate the implementation of a new system. The new system will likely inherit requirements and architecture from the existing systems [2].

Choose One (CO) Evaluate the existing systems, choose the one that is most satisfactory, and

discontinue all others. The chosen system will likely need to be evolved before it can fully replace the other systems.

The fourth option is to reuse parts from more than one existing system and re-assemble them into a new system, and we present two such types of merge, distinguished by the time required to implement them:

Instant Merge (IM) The existing components are rearranged with only minor modifications or development of adapters.

Evolutionary Merge (EM) Continue development of all existing systems towards a state in which architecture and most components are identical or compatible.

Selecting a strategy is naturally influenced by many factors. A reasonable starting point would be to early focus on questions and issues that could rule out some strategies. The two concerns we have found are:

Architectural Compatibility The notion of *architectural mismatch* is not new [1]. If the architectures of the existing systems are not very similar, *Instant Merge* can be excluded, and if the systems are very dissimilar also *Evolutionary Merge*. In reality, compatibility is not so easily categorized and must be assessed in each new situation. Compatibility issues found in the cases were: similarity of component roles and high-level structures [2]; data models (cases *F1*, *F2*, *F3*); similarity of underlying frameworks, i.e. how components are defined, for example ‘processes communicating via files’ (*F1*), certain hardware topologies (*C*), and identical development and deployment environments (*F3*). Similar ancestry and standards may imply a certain amount of compatibility (*C*, *D*, *F2*) [2].

Retireability Retiring a system may be considered more or less feasible, based on influences such as: investments made (*B*, *F1*, *F2*) and the satisfaction of various stakeholders: architects (*A*, *C*, *E1*, *F2*), users

and customers (*B, C, D, F2, F3*), management (*A, B, E2, F1, F2, F3*). Affection to ‘ones own’ system also influences the will to retire it. If only some or none of the systems are considered possible to retire, the strategies *Start from Scratch* and *Choose One* can be excluded. Retireability is typically re-evaluated given the resulting set of possible strategies, until an acceptable balance is found between the estimated cost of integration and the problems caused by retirement (*A, B, C, E2*).

Table 1 summarizes the exclusion of strategies based on these concerns (black denotes exclusion). Table 2 shows the results for *Architectural Compatibility* and *Retireability* in the cases, and Table 3 the resulting exclusion of strategies (black background) and the chosen strategy (circles).

3. Discussion and Future Work

The proposed scheme, with five strategies and two concerns to exclude strategies, may seem trivial and self-evident. It seems that some of the cases however were not aware of this, and unnecessarily spent time and energy without significant progress (cases *C* and *F2* in the tables). These concepts should therefore be useful for the software architect to focus analysis and discussions.

There are several research directions for the future. First, the strategies are not so distinct in reality and could be further refined; for example, although the overall strategy may be *Start From Scratch* or *Choose One*, some parts may still be reused from several existing systems. Second, we would like to further investigate the notion of (in)compatibility: further studying problems reported from practice would form a basis for how compatibility can be assessed, and also

suggest specific patterns or mechanisms to overcome incompatibilities. Third, we would like to see guidelines on when and how to involve different stakeholders in order to negotiate retireability and finally select a strategy.

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

4. References

- [1] 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.
- [2] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Reuse in Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of First International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2005.
- [3] 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.
- [4] 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)*, 2005.
- [5] Sewell M. T. and Sewell L. M., *The Software Architect's Profession - An Introduction*, Software Architecture Series, ISBN 0-13-060796-7, Prentice Hall PTR, 2002.

Table 1: The exclusion of possible strategies

		SFS	CO	EM	IM
Architectural Compatibility	Very high				
	Modest				
	No				
Retireability	All				
	Not all				
	None				

Table 2: Concerns per case

	Retireability	Compatibility
A	All	No
B	Not all (One)	No
C (initially)	None	Somewhat
C (final)	Not all (Either)	
D_{HMI}	Not all (One)	No
*D_{Server}	(?)	Somewhat (?)
E1	All	Somewhat
E2	Not all (One)	Somewhat
F1 (initially)	No	No
F1 (final)		
F2_{Pre}	All	No/Somewhat (?)
*F2_{2D}	All	Somewhat
F2_{Post}	All	No
*F2_{3D}	None	Somewhat
F3	All	No/Somewhat (?)

Table 3: Possible and desired strategies, per case

	SFS	CO	EM	IM
A	○			
B		○		
C'				○
C''		○		
D_{HMI}		○		
*D_{Server}	(?)	(?)	○	(?)
E1	○			
E2		○		
F1'			○	
F1''		○		
F2_{Pre}	○		(?)	
*F2_{2D}		○		
F2_{Post}	○			
*F2_{3D}			○	
F3	○		(?)	

Notes: For cases *C* and *F1*, there are two rows each to describe how decisions changed over time. Cases *D* and *F2* has been divided according to subsystem boundaries. An asterisk indicates that the integration is still at the planning stage.

Appendix B

Interviews

B.1 Case A

Date: 2005-11-21 08:15 - 09:45

1. What is your background? When did you join the company? What kinds of work are you involved in? What was your position in this project?

Interviewee studied in a joined electrical and computer engineering program specializing in electrical engineering. He joined the company in 1999 in corporate training program for Software Developers and was offered a job at Corporate Research. Before the project interviewee performed a competitor study for the product and become familiar with the working details, therefore he was selected as project leader on the assessment project. Research interest includes user interface design and usability.

2. Who were involved in the assessment? What were their backgrounds? How much time was spent on the assessment?

One full-time project leader (interviewee), 3 50

The people from corporate research where hired for two reasons: their experience with assessments and the fact that they are unbiased towards an internal development.

The first 2 months were spent on clearly stating the requirements, another 2 months to filter out unviable options reducing the possible options from 9 to 2. After this initial assessment the team was reduced to the project manager and the 3 developers. The remaining 6 months were spend on evaluating the 2 alternatives plus the option of developing the component in-house

3. What is the main system the project was concerned with? What kind of component were you looking for? Why were you looking for a component?

The system is a large scale control system including a GUI builder that allows for customisation of the control interface by the customer.

The sought after component had to be a replacement for the current GUI builder. The reason to look for a new component was that the current builder is based on technology that is no longer supported by the vendor and in order to guarantee the future-proofness of the main system it had to be replaced by a future-proof component.

The idea held by management was that acquiring a third party component as opposed to developing a new version in-house would be cheaper and have a better time-to-market. This assumption is part of the current trend in this industry. However the option to develop a component in-house was explicitly left open.

4. How did you formulate the requirements for the component to be selected? How complete and formal were these requirements? How were they documented?

The requirements were based on the user requirements for the current system and the system requirements derived from those. The new system should fulfil all the requirements of the current system. New requirements were gathered through informal discussions with product management and developers of the current system. However the requirements were at such a high-level that they were not directly usable for the assessment and the first 2 months were spend on refining the requirements though talks with product management and developers.

One of the major requirements was that the new component should fit in the existing system, without having to modify the system itself.

5. How did you make the initial selection of components to assess? What criteria did you use? What is the likelihood that this selection did select the component with the highest potential?

The initial select was made in a brainstorm with different stakeholders from both corporate research and the division. Input about the available components on the market was provided by the product managers based on information gathered at trade fairs. Out of the 9 products initially suggested 3 were immediately 'discarded' because they did not fit the basic requirements or the desired functionality was part of a much larger component with undesirable and irremovable features.

6. What characteristics of the component and the system do you consider when determining whether they are compatible? How did you come to these characteristics? Which characteristics where not include but should have been in retrospect?

Characteristics divided into 3 sections: technical, market and business.

The technical requirements consisted of requirements on what the builder should be able to do such as move, rotate, align, etc. and also included requirements on the fact that the editor had to be compatible with the hierarchical way the components to be represented are handled in the main product.

The main market requirement was a fast time-to-market.

Business requirements included cost of licensing, type of licensing, division of support responsibility for final product, priority of support by component company etc.

7. How do you obtain the information about the characteristics of the component necessary to determine the compatibility between the component and the existing system? Confidence?

Since the main requirements was that the product had to be future-proof and therefore based on the new platform technology from a vendor that is about to replace the older technology. The fact that the platform is relatively new means that none of the 2 vendors had a finished component, or even an Alpha version. Therefore no documentation, including requirements documents, existed for the new products in-development based on the new technology. Documentation did however exist for the old products but that was not considered useful since the underlying technology would change completely.

Therefore information was obtained from meeting with the companies, with both marketing people and developers. Also the assessment team were provided with binary prototypes of the product in development.

Information about the time-to-market was particularly hard to obtain because none of the components was close to be completed.

Although the component companies were very keen on selling their product or partnering with the company the information they provided was considered realistic with a considerable confidence.

Furthermore for both components and the internal development prototypes were created to obtain additional information.

Finally a standardised survey was filled out by the component companies inquiring about their financial situation, including size, and past revenues in order to establish whether the component company is financial stable.

8. Once the characteristics are known how do you compare them with the existing system? Can they differ and still be compatible? If so in what way can they differ? Characteristics were compared during meeting between developers of the system and the component.
9. What trade-offs were encountered during the assessment? How were the trade-off decisions made? Which characteristic is most important?

Since both the components were in a very early stage of development, one of the main trade-offs to be made was whether to use a third-party component or internally develop a new component. The major advantage of selecting a third-party component as this early stage of development is that the development can be influenced in favour of compatibility with the existing system. However this also causes two major disadvantages, one is that core knowledge of part of a product would lay outside the own company, the second is the risk involved in committing to an unfinished component.

Another problem is that the structure of the existing product is rather unique and that the third-party components did not comply with this structure, making it necessary to considerably modify (> 25% modification of code) the components or require a separate product line from the companies. Such a construction causes

all kind of complications regarding support agreements and responsibilities for problems.

Considering this the common advantages of using third-party components did not exist in this case.

10. What other aspects influence they choice of a component?

The developers of the division were in favour of developing the component in-house, while the product management was in favour of a third-party component. These different views caused communication problems between management and developers.

11. Which component was selected? And for what reasons?

Four of the 6 component did not implement the functionality themselves but provided a class library that could be used to implement the functionality. However none of them were written using the desired platform and therefore not considered future-safe and discarded. This left 2 component under consideration and the possibility of developing the component in-house.

Neither of the two components under consideration fulfilled the technical requirements completely, although one more than the other. However both would require significant ($> 25\%$) modification. Neither of the components was in a state that is was, or would be in the short term, mature enough to be used.

Therefore both components were rejected and it was decided to develop the component in-house.

12. How correct were the characteristics found during the selection process? Did the integration encounter unforeseen incompatibilities? Where they caused by characteristics that were not considered or by characteristics for which the information was incorrect or incomplete?

N/A

13. What is the current status of the integration effort?

The company is about to release an internal alpha after 11 months of development. Final release is expected in another 5-6 months.

However in the mean time the vendor has announced yet another big upgrade to the underlying technology requiring a partial rewrite of the component to incorporate the new technology.

14. What is your evaluation of the assessment and consequent integration process?

The challenge was not so much technical as it was to get a common view between management and developers about the product. The assessment succeeded in getting this common view and was therefore a success.

One of the success factors was involving the developers in the assessment process.

B.2 Case B

Date: 2005-12-23

1. What is your background? When did you join the company? What kinds of work are you involved in? What was your position in this project?

Interviewee has been working as software architect for a consulting company for over 10 years and in that capacity has been involved in numerous software projects in financial, energy, transport industry and public sector. Before the start of the project the interviewee had been involved in a large project for the financial institution in question. So he is aware of the internal structure and infrastructure of the company.

2. Who were involved in the assessment? What were their backgrounds? How much time was spent on the assessment?

The assessment was performed in 1999. Besides the interviewee a group of 4 to 9 other architects, all from different consulting companies, made up the architecture team. The team was lead by project leader from the financial institution. The different members of the team all had architecture knowledge and experience. This type of architecture team with exclusively external consultants is not uncommon in this domain.

The total assessment, including drafting requirements, initial selection, final selection, and prototyping, took approximately 2 person years.

3. What is the main system the project was concerned with? What kind of component were you looking for? Why were you looking for a component?

The goal of the project, called the Multi Channel Platform (MCP), was to provide an infrastructure for integrating services in a uniform way in the business of retail banking (high volume, high availability). This in order to support the business goal of providing information in a multi-channel fashion, e.g. through a bank office and the internet. In order for that to work there needs to exist a single customer view between all (mainframe and other) systems containing and handling data. Hence the need for a uniform way of (re)integrating them. This is nowadays referred to as an Enterprise Service Bus.

The scope of the project, i.e. which systems were going to be (re)integrated using the Enterprise Service Bus, was not clearly defined at the outset of the project.

The field of messaging busses, which is a major part of an Enterprise Service Bus, was considered mature and building such a complex piece of middleware by the company itself was considered undesirable from a financial and maintenance point of view.

4. How did you formulate the requirements for the component to be selected? How complete and formal were these requirements? How were they documented?

The requirements were formulated by the architecting team with the help of people within the financial institution. The process was a fully “featured” requirements

gathering process which were organized according to the categories defined in ISO 9126. The total number of requirements was several 100. Every requirement was prioritized which turned out to be one of the more difficult aspects of the process. The metrics in the requirements were estimates based on information from the institution.

The requirements process was done very thorough in accordance with the culture in the institution.

The requirements did not explicitly state which systems were going to be (re)integrated using the Enterprise Service Bus, in other words the scope of the project was not clearly defined.

5. How did you make the initial selection of components to assess? What criteria did you use? What is the likelihood that this selection did select the component with the highest potential?

The initial selection was made based on the experience of the architecture team and possibly searching the internet. This resulted in a long list of approximately 10 products including BEA Tuxedo, DEC MessageQ, IBM MQ Series, IONA Orbix, and Tibco Rendezvous.

The long list was reduced to a short list based on the assessment of the architectural of whether the component was suitable for the task and whether the company that behind the component would be a stable and reliable partner. Especially the latter was very important for the financial institution to make sure that the selected product was “future-proof”.

The short list contained 3 products: BEA Tuxedo, IBM MQ Series, and Tibco Rendezvous.

6. What characteristics of the component and the system do you consider when determining whether they are compatible? How did you come to these characteristics? Which characteristics were not included but should have been in retrospect?

The systems that were to be integrated using the Enterprise Service Bus used both asynchronous and synchronous communication. In order for the ESB to function properly it would need to support both types of communication. This did not necessarily mean that the messaging bus component needed to support both, however if it didn't an extra layer would need to be added to implement the missing feature.

Another characteristic which was considered important was installability/configurability of the component.

7. How do you obtain the information about the characteristics of the component necessary to determine the compatibility between the component and the existing system?

Initially the suppliers on the short list and several from the long list were invited to give a presentation to inform the architecture team about the capabilities of their product.

Characteristics of the components were obtained by sending the suppliers an extensive questionnaire with open questions based on the list of requirements. The suppliers responded in detail to the questionnaire because the financial institution was potentially a very lucrative customer. The answers to the questionnaire were considered a commitment by the supplier to actually fulfil their claims.

The answers to the open questions were scored 1 to 5 (highest) by the architecture team and assigned to the respective requirements. After that the scores were added up weighing them using the requirement's priority. The results of this scoring can be influenced by flaws in the translation from the textual answer to a score and by the incorrect assignment of priorities to requirements. However because the large number of requirements these "flaws" were assumed to be "compensated".

The answers to the questionnaire were then discussed with the suppliers.

Another source of information were the references to other customers, provided by the three suppliers, whom were interviewed about their experience with the component. This provided some additional information, however since only a limited time was available from the references this process was not extremely detailed.

8. Once the characteristics are known how do you compare them with the existing system? Can they differ and still be compatible? If so in what way can they differ?

There was not really an existing system in the sense of a single, clearly identifiable component that provided the functionality to be replaced. In stead ad-hoc integrations existed between systems.

9. What trade-offs were encountered during the assessment? How were the trade-off decisions made? Which characteristic is most important?

Main trade-off issues included handling of synchronous communication, maintenance, and reliability of vendor. The trade-offs were made based on the priorities assigned during the requirements gathering.

As will be clear from the answer to the next question reliability of vendor was the most important characteristic.

10. What other aspects influence they choice of a component?

After the recommendation to select BEA Tuxedo was made the financial institution's management indicated that IBM should be given another chance, clearly indicating their preference for this supplier. The main reason for this was the existing relations with IBM and the fact the IBM's PR management and responsiveness was better than that of BEA giving a more reliable impression, which is a very important quality to a financial institution since a commitment to a component is usually very long term.

The architecture team re-invited IBM to give their presentation and changed their recommendation to IBM MQ series.

11. Which component was selected? And for what reasons?

The component that was recommended by the architecture team was BEA Tuxedo because it could handle both synchronous and asynchronous communication well,

because its solutions were considered more elegant, and because ease of installing, configuring, and maintaining it.

However because of other aspects mention in question 10 IBM MQ Series was eventually selected. MQ Series was the second choice and was not far behind Tuxedo from a technical point of view, however the main disadvantage was that it did not support synchronous communication.

12. How correct were the characteristics found during the selection process? Did the integration encounter unforeseen incompatibilities? Where they caused by characteristics that were not considered or by characteristics for which the information was incorrect or incomplete?

After the selection of the component a proof-of-concept/prototype was made so test a large number of requirements. During the building of the prototype and later on during the building of the system approximately 2 IBM engineers were employed. There was also contact with the IBM development lab that developed MQ series.

During the prototyping no unforeseen properties, including incompatibilities, were found. What was found was the expected difficulties with the complex API and the complexity of the configuration.

During the building of the system it was found that building a synchronous communication layer on top of MQ Series was not viable and the idea was abandoned. This meant that it was more difficult to use the bus to integrate systems that work with synchronous communication, see also scope change in next question.

13. What is the current status of the integration effort?

The project was completed successfully, however the scope was scaled back from a company wide Enterprise Service Bus to a messaging bus to integrate the backend mainframes with the business and application tiers.

14. What is your evaluation of the assessment and consequent integration process? Positive and negative points? Main success/failure factors?

The interviewee was not involved in a formal evolution of the project. However he would grade the entire process, minus the management decision to select IBM, with 8 out of 10. Especially thorough was the requirements process and the following selection process, indicated by the fact that the requirements list was suitable for (partial) reuse in several later projects.

A less successful point was the attempt to build synchronous messaging functionality including marshalling on top of the already complex API. This part was eventually abandoned.

In order for the message bus to function as ESB several functions needed to be added, these were added, in retrospect, although that functionality wasn't available at the time the decision was made, it would have been better to use a commercial solution for this.

The conclusion by the interviewee is that at the time of the project there were no possibilities to improve it, however later projects learned from the experiences with assessing products in this project.

Appendix C

Questionnaire

This appendix contains a facsimile questionnaire as it was sent to the respondents. The original document contained interactive elements allowing the respondent to fill the questionnaire using Adobe Acrobat and returning it by pressing Submit.

Questionnaire

The goal of our research is to improve the methods used for selecting third-party software components to be used in software systems. Third-party software components are self-contained pieces of software developed by an external party. The components can be either commercial or open-source. In order to do improve current methods we are interested in the current state of practise, therefore we ask you to fill in this questionnaire about the use of third-party components by you and your company.

This questionnaire contains between 11 and 55 questions and should take between 10 and 30 minutes to fill out depending on your answer to certain questions. The answers to the questions will be treated confidentially and used exclusively for this research.

The questionnaire can be submitted electronically by email or in printed for using postal mail. The text at the last page of the questionnaire will guide you through the submission.

We very much appreciate you filling out this questionnaire even if you feel you have limited experience working with software components.

General

The following questions are intended to provide us with a general background of you and the company you work for.

1. How many years have you been working with software development?	<input type="radio"/> 0 – 1 year <input type="radio"/> 1 – 2 years <input type="radio"/> 2 – 5 years <input type="radio"/> 5 – 10 years <input type="radio"/> 10 – 20 years <input type="radio"/> 20+ years <input type="radio"/> None
2. How many years have you been in your current position?	<input type="radio"/> 0 – 1 year <input type="radio"/> 1 – 2 years <input type="radio"/> 2 – 5 years <input type="radio"/> 5 – 10 years <input type="radio"/> 10 – 20 years <input type="radio"/> 20+ years

<p>3. Which function(s) describe your current position best? (Multiple answers possible)</p>	<p><input type="checkbox"/> Requirements analyst</p> <p><input type="checkbox"/> Architect</p> <p><input type="checkbox"/> Designer</p> <p><input type="checkbox"/> Programmer</p> <p><input type="checkbox"/> Tester</p> <p><input type="checkbox"/> Manager</p> <p><input type="checkbox"/> Other: <input style="width: 150px; height: 20px;" type="text"/></p>
<p>4. How many people work in software development in your company?</p>	<p><input type="radio"/> 1 – 10 developers</p> <p><input type="radio"/> 10 – 50 developers</p> <p><input type="radio"/> 50 – 100 developers</p> <p><input type="radio"/> 100 – 500 developers</p> <p><input type="radio"/> 500 – 1000 developers</p> <p><input type="radio"/> 1000+ developers</p>
<p>5. What kind of domains is your company active in? (Multiple answers possible)</p>	<p><input type="checkbox"/> Real-time systems</p> <p><input type="checkbox"/> Business systems</p> <p><input type="checkbox"/> Embedded systems</p> <p><input type="checkbox"/> Industrial systems</p> <p><input type="checkbox"/> Office automation</p> <p><input type="checkbox"/> Consulting</p> <p><input type="checkbox"/> Other: <input style="width: 150px; height: 20px;" type="text"/></p>
<p>6. What percentage of the software projects in your company is completed within <i>twice</i> the budget and within <i>twice</i> the time originally planned?</p>	<p><input type="radio"/> 0 – 10%</p> <p><input type="radio"/> 10 – 25%</p> <p><input type="radio"/> 25 – 50%</p> <p><input type="radio"/> 50 – 75%</p> <p><input type="radio"/> 75 – 100%</p> <p><input type="radio"/> Unknown</p>
<p>7. Do you use third-party components in your software?</p>	<p><input type="radio"/> Yes, <i>please continue with question 11</i></p> <p><input type="radio"/> No, <i>please continue with question 8</i></p>

Please continue either with question [11](#) or [8](#) depending on your answer to question 7.

8. Do you use internally developed components in your software?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
9. Why do you not use third-party components? (Multiple answers possible)	<input type="checkbox"/> Too expensive <input type="checkbox"/> No components available <input type="checkbox"/> Quality not good enough <input type="checkbox"/> Conflicts with company policies <input type="checkbox"/> Bad experience <input type="checkbox"/> Other: <input style="width: 150px; height: 40px;" type="text"/>
Comments: <input style="width: 750px; height: 80px;" type="text"/>	

10. Have you considered using third-party components?	<input type="radio"/> Yes, <i>please continue with question 12</i> <input type="radio"/> No, <i>please continue with question 65</i>
--	---

Please continue either with question [12](#) or [65](#) depending on your answer to question 10.

Answer this question if you use third-party components:

11. On average, how much of the end product is consists of third-party components?	<input type="radio"/> 0 – 10% <input type="radio"/> 10 – 25% <input type="radio"/> 25 – 50% <input type="radio"/> 50 – 75% <input type="radio"/> 75 – 100%
Comments: <input style="width: 750px; height: 80px;" type="text"/>	

Specification

This section contains questions regarding to what extend you write a component specification or component requirements for the component(s) you are looking for.

12. In general do you start writing a detailed specification before searching for available components?	<input type="radio"/> Yes, please continue with question 13 <input type="radio"/> No, please continue with question 19
--	---

If you answered no to the previous question please jump to question [19](#), otherwise please continue with the next question.

13. Do the requirements include non-functional requirements such as performance, security, reliability, etc?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
14. Are the requirements prioritized?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
15. How much of the total development time do you, on average, spend on writing the <i>specification</i>?	<input type="radio"/> 0 – 5% <input type="radio"/> 5 – 10% <input type="radio"/> 10 – 20% <input type="radio"/> 20 – 50% <input type="radio"/> 50 – 100% <input type="radio"/> Unknown
16. Are the requirements evolved/refined during the assessment?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
17. In case of no suitable component, is it possible to modify the requirements?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
18. Do you feel having no or only general requirements would make the search for a component easier?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
Comments: <div style="border: 1px solid black; height: 80px; width: 100%;"></div>	

Please continue with question [26](#).

Please only answer the following questions if you do **not** write a specification before searching for components:

19. Do you have a set of basic requirements before starting to search for suitable components?	<input type="radio"/> Yes, please continue with question 20 <input type="radio"/> No, please continue with question 24
20. Do you document these basic requirements?	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
21. Do the basic requirements include non-function requirements such as performance, security, reliability, etc?	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
22. Do you refine the requirements based on the components you encounter?	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
23. Do you use refined requirements for excluding components that are not suitable?	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
24. Why do you <i>not</i> write a (detailed) requirements specification before searching for a component?	<input type="checkbox"/> Not enough time <input type="checkbox"/> Requirements not known <input type="checkbox"/> Other: <div style="border: 1px solid black; width: 150px; height: 80px; display: inline-block; vertical-align: middle;"></div>
25. Do you feel having (detailed) requirements would make the search for a component easier?	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
Comments: <div style="border: 1px solid black; width: 100%; height: 100%;"></div>	

Please continue on the next page.

Assessment

This section contains questions about the assessment of the components. The first set of questions relates to how you gather information about components.

26. Where do you obtain information about potential components?	
a) Trade fairs or conferences	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
b) Professional literature (E.g. magazines, conference proceedings, and books)	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
c) Word-of-mouth	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
d) Internet	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
e) Other source of information:	<div style="border: 1px solid black; height: 100px; width: 100%;"></div>
27. What kind of documentation is provided with the component?	
a) Architectural description	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
b) Design documents (e.g. UML)	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always
c) Usage examples	<input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always

35. What other sources of information do you use?	
a) References provided by the vendor	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
b) Information provided by colleagues	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
c) Information from commercial reviewers	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
d) Other	<div style="border: 1px solid black; height: 100px; width: 100%;"></div>
36. How much of the total development time do you, on average, spend on gathering information?	<input type="radio"/> 0 – 5% <input type="radio"/> 5 – 10% <input type="radio"/> 10 – 20% <input type="radio"/> 20 – 50% <input type="radio"/> 50 – 100% <input type="radio"/> Unknown
37. How good do you consider yourself/your company to be at gathering information about components in your company?	<input type="radio"/> Bad <input type="radio"/> Average <input type="radio"/> Excellent
Comments:	
<div style="border: 1px solid black; height: 150px; width: 100%;"></div>	

Testing

The next questions relate to testing of third-party components.

46. Do you test third-party components before integrating them into your system?	<input type="radio"/> Yes, please continue with question 47 <input type="radio"/> No, please continue with question 59
---	---

Please only continue with the next question if you do test third-party components, otherwise jump to question [59](#).

47. Do you use prototyping to determine whether or not a component is suitable?	<input type="radio"/> Yes, please continue with question 48 <input type="radio"/> No, please continue with question 54
48. How often do you build prototypes?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
49. When you build prototypes do you build them for each component individually?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
50. When you build prototypes and when using multiple components, do you build a prototype to see how they interact?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
51. Does the vendor of the component assist in prototyping?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
52. Does the vendor of the component provide assistance free of charge?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
53. Does prototyping provide you with information about a component not included in the information gathered before?	<input type="radio"/> Never <input type="radio"/> Seldom <input type="radio"/> Sometimes <input type="radio"/> Often <input type="radio"/> Always
Comments:	<div style="border: 1px solid black; height: 100px;"></div>

Please continue with question on the next page.

Please answer the questions below when you **do** test third-party components.

<p>54. How much of the total development time do you, on average, spend <i>testing</i> (and <i>prototyping</i>)?</p>	<p> <input type="radio"/> 0 – 5% <input type="radio"/> 5 – 10% <input type="radio"/> 10 – 20% <input type="radio"/> 20 – 50% <input type="radio"/> 50 – 100% <input type="radio"/> Unknown </p>
<p>55. What (other) kinds of testing of third-party components do you use?</p>	<div style="border: 1px solid black; height: 100px;"></div>
<p>56. Does testing (and/or prototyping) change your opinion about the suitability of a component?</p>	<p> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always </p>
<p>57. Have you ever come to the conclusion after testing (and/or prototyping) that, although a component seemed promising, it does not meet your requirements?</p>	<p> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Never Seldom Sometimes Often Always </p>
<p>58. How good do you consider yourself/your company to be at testing (and prototyping) components?</p>	<p> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> <input type="radio"/> Bad Average Excellent </p>
<p>Comments:</p> <div style="border: 1px solid black; height: 100px;"></div>	

Please continue with question [59](#).

65. Would you like to add additional information or comments?

The answers to the questions will be treated confidentially and used exclusively for this research. The answers, or references to you or your company will **not** be shared with third parties in anyway without your explicit consent.

We would like to contact you in the future regarding this type of research, however if you do not want to be contacted again please check the box below:

Please do not contact me again regarding this type of research.

Submit

When clicking the submit button Adobe Acrobat will attempt to compose a mail with the answers to the questionnaire using your preferred email program. The answers are attached to the mail in an XML file. The mail does not contain any information other then the answers to the questions and the email address you send it from.

Print

Alternatively you can click the print button. This will print out a copy of the questionnaire including the answers on a printer of your choice. Please send the printed questionnaire to:

*Mälardalens högskola
Laurens Blankers
Box 883
721 23 VÄSTERÅS
Sweden*

Thank you very much for filling out this questionnaire!

Laurens Blankers

Appendix D

Questionnaire results

The raw results of the questionnaire (see appendix C) are included here. Some of the comments by respondents have been removed or abbreviated in order to guarantee respondent's anonymity.

Resp.	Question 4	5 RT	Bus	Em	Ind	Office	Cons	Other	6	7	8
1	1-10		Yes		Yes		Yes		75-100	Yes	5
2	1000+	Yes	Yes	Yes		Yes	Yes		75-100	Yes	
3	1000+		Yes						75-100	Yes	
4	100-500			Yes					NA	No	3
5	1-10		Yes		Yes				25-50	Yes	
6	1000+	Yes	Yes		Yes	Yes	Yes		0-10	Yes	4
7	1000+			Yes					NA	Yes	
8	1-10	Yes		Yes	Yes		Yes		NA	Yes	
9	1000+	Yes		Yes	Yes				75-100	Yes	
10	1000+	Yes		Yes	Yes				75-100	Yes	
11	10-50				Yes				NA	Yes	
12	1-10		Yes						NA	Yes	
13	1000+	Yes	Yes	Yes	Yes	Yes	Yes		0-10	No	4
14	1-10			Yes			Yes		75-100	Yes	
15	10-50							Yes	25-50	Yes	
16	1000+	Yes		Yes	Yes				75-100	Yes	
17	10-50			Yes	Yes				NA	Yes	
18	50-100	Yes		Yes	Yes				75-100	Yes	
19	500-1000	Yes		Yes					NA	Yes	5
20	1-10		Yes	Yes		Yes	Yes		0-10	Yes	
21	1-10						Yes		75-100	Yes	
22	500-1000	Yes	Yes	Yes	Yes	Yes	Yes		NA	Yes	
23	10-50	Yes	Yes	Yes	Yes	Yes	Yes		75-100	Yes	

Resp.	Question 9		Expensive	No needed	Quality	Conflict	Experience	Other	10	11	12	13	14	15
	Expensive	No needed												
1								Yes	0-10	No				
2									50-75	No				
3									25-50	No				
4			Yes					No						
5									10-25	No				
6									50-75	Yes	5	4	5	
7									50-75	No				
8									10-25	No				
9									10-25	Yes	3	4	5	
10									10-25	No				
11									0-10	No				
12									10-25	No				
13		Yes	Yes	Yes	Yes	Yes	Yes	No						
14										Yes	4	3	5	
15									10-25	No				
16									10-25	No				
17									0-10	No				
18									25-50	No				
19									50-75	No				
20									0-10	No				
21									0-10	Yes	3	4	10	
22									50-75	Yes	5	4	10	
23									25-50	No				

Resp.	Question 16	17	18	19	20	21	22	23	24 Time	Unknown	Other
1				Yes	2	5	3	4			Yes
2				Yes	5	3	4	2			Yes
3				Yes	3	4	4	4			Yes
4											
5				Yes	5	5	4	4			Yes
6	4	2	3								
7				Yes	3	5	4	3		Yes	
8				Yes	5	4	4	2			Yes
9	4	3	5								
10				Yes	5	4	2	3		Yes	
11				Yes	3	2	3	3		Yes	
12				Yes	4	3	1	1			Yes
13											
14	4	3	3								
15				Yes	4	4	3		Yes	Yes	Yes
16				Yes	4	2	4	4		Yes	
17				Yes	4	3	3	4		Yes	
18				Yes	5	4	5	2			Yes
19				Yes	5	5	5	3		Yes	
20				Yes	4	3	4	4			
21	4	4	1								
22	4	3	2								
23				Yes	5	2	2	2	Yes	Yes	

Resp.	Question 28	29	30	31	32	33	34	35 Ref	Col	Com	Other
1	3	3	3	Yes	2	4	3	2	4	1	
2	3	2	3	Yes	5	4	3	3	4	3	
3	3	2	4	Yes	4	4	3	4	4	3	Internet
4											
5	2	2	4	Yes	4	3	2	2	4	2	Internet
6	3	2	3	Yes	4	4	3	3	4	2	
7	2	2	2	Yes	4	4	2	2	4		
8	4	3	4	Yes	4	4	4	4	4	3	
9	3	3	4	Yes	4	4	3	3	5	3	
10	2	3	3	Yes	4	2	2	3	3	2	
11	3	3	3	No				1	3	2	
12	3	3	4	Yes	2	2	2	2	4	4	
13											
14	3	3	3	Yes	3	3	2	3	4	2	
15	4	2	5	Yes	5	2	3	1	4	2	
16	3	3	4	Yes	4	4	3	4	4	2	
17	2	3	3	No							
18	3	3	3	Yes	4	4	4	2	3	4	
19	3	2	2	Yes	5	4	3	2	3	1	
20	3	3	3	Yes	3	3	3	3	3	3	
21	3	3	3	Yes	3	3	3	3	4	3	
22	4	3	3	Yes	5	5	3	5	5	4	
23	3	2	3	Yes	4	4	4	2	3	2	

Resp.	Question	36	37	38	39	40	41	42	43	44	45	46	47
1		NA	3	No	1		5	4		0-5	3	Yes	Yes
2		NA	3	Yes	2	[6]	4	3	Users	NA	3	Yes	Yes
3		0	3	Yes	1		4	5	Legal	0-5	3	Yes	Yes
4													
5		10	4	No	1		4	1		5-10	3	Yes	No
6		5	4	Yes	3	[7]	4	3	Financial	0-5	4	Yes	Yes
7		20	4	Yes	4	[8]	3	3		10-20	4	Yes	Yes
8		10	3	No	2		2	2		0-5	3	Yes	Yes
9		5	4	No	2		4	4	Management	5-10	4	Yes	Yes
10		10	3	No	1		4	1		0-5	3	Yes	Yes
11		20	3	No	3		2	2		5-10	3	No	
12		0	3	No	1		4	1	Legal	0-5	3	Yes	Yes
13													
14		0	4	No	1		2	3	Business	0-5	3	Yes	Yes
15		0	3	No	1		1	2	Financial	0-5	2	Yes	Yes
16		10	3	No	1		4	4	Supply	0-5	3	Yes	Yes
17		10	2	No	1					10	3	Yes	Yes
18		10	3	No	3	[9]	4	4	Business	5-10	4	Yes	Yes
19		10	4	Yes	4	[10]	4	5	Supply	5-10	3	No	
20		5	4	Yes	3		2	1		5-10	3	Yes	Yes
21		10	3	No	1		1	1		0-10	3	Yes	Yes
22		20	4	Yes	4	[11]	5	4	Business	10-20	3	Yes	Yes
23		10	3	No	1		2	3	Hardware	0-10	3	Yes	Yes

Resp.	Question 48	49	50	51	52	53	54	55	56	57	58	59
1	5	3	3	1	4	4	0		4	4	4	3
2	4	2	4	3	2	5	NA		3	3	3	5
3	4	4	4	2	3	5	0	[12]	4	3	3	3
4												
5							10	[13]	4	4	4	4
6	4	3	4	2	4	4	20	[14]	3	3	4	3
7	3	3	2	2	3	4	10		4	4	4	4
8	4	3	4	1	2	3	5		3	2	2	5
9	3	2	4	2	2	4	10		4	3	5	4
10	4	2	4	1	2	4	5		3	3	3	3
11												3
12	5		5	1		4	0	[15]	4	3	3	4
13												
14	4	2	4	3	4	3	10		3	3	3	3
15	4	5	2	1	2	5	5		3	1	1	3
16	4	2	4	4	4	4	0		4	3	4	3
17	4	3	4	4		2	10		3		3	3
18	4	3	3	2	5	5	10		3	3	4	3
19												5
20	5	4	4	1	2	4	10		4	3	3	3
21	5	4	3	1	1	2	5		3	3	3	3
22	4	3	3	4	4	4	20		3	3	4	4
23	3	2	4	2	4	5	20		3	2	4	3

Resp.	Question	60	61	62	63	64
1	3	2	4	5	4	
2	3	2	4	NA	4	
3	3	2	4	5	3	
4						
5	3	2	4	10	3	
6	3	4	5	10	5	
7	3	3	3	20	4	
8	3	3	3	5	3	
9	3	2	3	10	2	
10	2	3	4	10	4	
11	1	2	3	0	3	
12	4	4	3	10	3	
13						
14	3	2	3	5	3	
15	5	4		5	3	
16	3	3	2	5	3	
17	2		3	5	3	
18	2	2	3	NA	3	
19	3	3		50	4	
20	3	3	5	0	5	
21	3	3	3	10	4	
22	3	3	4	50	5	
23	3	1	3	10	4	

- 1 approached by salespeople
- 2 Industry analysts (e.g. Gartner), Vendor presentations and briefings specific to our organization, User groups
- 3 Developers' Forums
- 4 Existing relations
- 5 Direct contact with vendors.
- 6 cost/benefit analysis, e.g. CBAM
- 7 Our quality management system [...]
- 8 [...] Checklist for Third Party components
- 9 [...]
- 10 checklists
- 11 Guidelines and methods do exist but they are not always easy to find when needed nor very well kept up-to-date.
- 12 Commercial testing of the vendor. Strategic assessment.
- 13 Unit testing, Integration testing , Load and Stress testing to test non functional characteristics
- 14 Self-created components, existing test-suites (both self-created and -the-shelf).
- 15 what do you mean?