ENTERPRISE APPLICATION INTEGRATION

An Investigative Case Study

Kristoffer Renholm

KTH Electrical Engineering

Degree project in
ICS
Master Thesis
Stockholm, Sweden 2011

XR-EE-ICS 2011:022

ENTERPRISE APPLICATION INTEGRATION: AN INVESTIGATIVE CASE STUDY

Kristoffer Renholm

A Master Thesis Report written in collaboration with the
Department of Industrial Information and Control Systems
Royal Institute of Technology
Stockholm, Sweden

October 2011

Abstract. Enterprise systems and applications are getting more and more complex. In 30 years computing has transformed from big single machines to highly heterogeneous systems. Businesses are today depending on a super-sized mixed bag of applications that have been introduced over the years. Businesses have much to gain by obtaining a holistic view of their data and integrating business processes that are spread over different systems.

This master of thesis explores critical success factors (CSF) for enterprise application integration (EAI) from the perspective of developing new application into such environment. Based on current research in the field of EAI, and experiences gained from a real-life scenario the report argues for the importance of integration and what the most influential factors for success are.

Among many significantly important factors, two main trends emerge as especially important: developer guidance and documentation, and integration functionality. The former leads to success by focusing on developer productivity through the use scorecards, principles and guidance, well-specified interfaces, and skilled EAI employees; and the latter by extending integration functionality to include many-to-many integration strategies and workflows to model the business.

Keywords. Enterprise Application Integration. Application Development. Case study. Critical Success Factors. Application Integration.

Acknowledgements

Waldo Rocha Flores – Master Thesis Supervisor, ICS KTH

Mikael Harrén – Supervisor Case Study

Carlos Lo-Iacono – Supervisor Case Study

Table of Contents

1 INTRODUCTION 4
1.1 Background 4
1.2 Purpose 4
2 ENTERPRISE APPLICATION INTEGRATION 5
2.1 Definitions and Characteristics 5
2.2 Driver and Benefits 5
2.3 Functionality 6
2.4 Critical Success Factors 8
2.5 Reference Framework 10
3 METHOD 12
3.1 Literature Study 12
3.2 Collecting The Evidence 13
3.3 Analysis 14
4 EMPIRICAL DATA 15
4.1 Case Details 15
4.2 Organisation 16
4.3 Environment 16
4.4 Functional Analysis 18
4.5 Implementation 19
5 ANALYSIS AND DISCUSSION 23
6 CONCLUTIONS 29
7 REFERENCES 31
Appendix A – Prototype class diagram 32

1 INTRODUCTION

1.1 Background

Enterprise systems and applications are getting more and more complex. In 30 years computing has transformed from big single machines to highly heterogeneous systems. Businesses are today depending on a super-sized mixed bag of applications that have been introduced over the years: best of breed applications that are pre-packaged and bought "of-the-shelf", legacy application not always desirable to replace, custom made applications meeting specific business needs, and database management systems. [1] This is mainly due to two reasons. First, because businesses often rushed in new systems without foresight into future needs. Secondly, IT decisions were made at the department level, resulting in different solutions and technologies for each department across the enterprise.

Today businesses are better at more holistically procuring software. The situation will however persist as businesses continue to need tailor-fit applications, use legacy applications, and keep buying packaged best-of-breed applications. [2] Meanwhile focus is shifting from introducing new systems to leveraging existing ones. [1] There are many examples of this: e-commerce applications that need to interact closely with many different systems, business intelligence solutions that gather data from across the enterprise. Yet another is the need to leverage existing systems in mergers and acquisitions when two IT environments merge. [2]

By integrating existing systems businesses can both differentiate themselves and stay competitive. First, they are able to do business more efficient and flexible. Second, respond faster to both internal and external pressure for change, e.g. the above-mentioned e-commerce. Third, integration can provide a competitive advantage when differentiating the infrastructure and using systems more efficiently (e.g. value-added services such as customization and shipping information). [2] [3] [4]

An attractive offer for businesses to pursue the benefits of integration is Enterprise Application Integration (EAI). It is a broad concept covering tools and techniques, as well as business processes and organisational transformations to modernize, consolidate, integrate, and coordinate systems. [1] Existing systems can be leveraged in a multitude of processes while presenting a unified view of the business to clients and partners. [2]

1.2 Purpose

The broad concept of EAI provides practitioners with many tools and techniques that can help to succeed with the daunting task of integration. In bordering research and in other industries, the use of critical success factors (CSF) has helped guide leaders navigate to successful solutions. Similar research has been carried out on EAI investigating what factors are important for successful integration projects. Results are yet largely in the makings and much of the prior research have focused on large change project with company-wide rollouts of integration. Few have investigated what makes introduction of new applications into integrated environments successful. Yet new business demands and technology opportunities increases the need to develop new application. This master thesis will thus, based on prior research and a case study, investigate the questions

- What critical success factors are important for the development and integration of new applications into integrated businesses?

1.3 Report Structure

This report is structured in four sections. In section 2, the theory is presented with further descriptions and definitions of enterprise application integration. This is followed by a description in section 3 of the methods that were used to conduct the study and gather data. The gathered empirical

data are then presented in section 4, and later analysed in section 5 together with the presented theory. Finally section 6 summarises the report and presents the final conclusions.

2 ENTERPRISE APPLICATION INTEGRATION

This chapter continues from the given background by first further motivating EAI and describing the techniques and tools available with a peek into service oriented computing. More focus will however lie on the critical success factors presented from literature, and followed by a framework to base the case study on. As foundation for discussing the case study environment, given are also a brief introduction component-based development.

2.1 Definitions and Characteristics

There are numerous EAI definitions in literature, all similar with shifting focusing between business and technology. One commonly used are "the plans, methods, and tolls aimed at modernising, consolidating, integration and coordinating the computer applications within an enterprise." [1] [3] Yet another that captures both technical and business are "the integration of application that enables information sharing and business processes, both of which result in efficient operations and flexible delivery of business services to the costumer" [2]. Essentially, and common to the both definitions, is the focus on integration that should lead to benefits and competitive advantage for the business.

To make a distinction from general information system (IS) projects, EAI projects differ in a variety of ways. First, and not that striking, are the focus on leveraging existing functionality and business processes. IS projects have instead focused on developing completely new applications. Second, it requires a more upfront planning by the interaction with multiple systems or across departments. Third, and since EAI can span different departments, there are often no single project owner, and usually multiple stakeholder groups in contrast to the single project owner and the one stakeholder group of IS projects. [1]

2.2 Driver and Benefits

There are many drivers of a more integrated business environment. First is globalisation and the new way companies do business introduced by the Internet. Internal systems and processes now need to provide functionality to and be available from web applications. Businesses become competitive by reducing their time-to-market for new offerings by using existing applications and knowledge. [2] [3] With EAI it is possible to integrate different system and business processes throughout the enterprise, enabling it to more casual adopt new business processes as needed. [3]

General competitive benefits also acts as a driver of integration. First, businesses can profit from operational benefits, including reduced cost of running, managing and maintaining the IT infrastructure. Second there are managerial benefits with more organised business processes and understanding how to control them. Third there are technical benefits involving flexible, manageable and maintainable IT infrastructure, reduced redundancy in both data and systems, and cheaper implementations. Fourth and last there are also organisational benefits as it allows organisations to do business more efficient, e.g. manual tasks be reduced or eliminated when processes are getting fully automated and integrated. [4]

Yet another driver of integration is rise of mergers and acquisitions. In each event when two or more completely different IT-environment meet, integration is need as businesses wants to expand their offerings while reducing costs and rationalise by eliminating redundancies. [2]

Benefits cannot however be achieved unchallenged. Today businesses have arrived to a situation with highly heterogeneous and overlapping systems throughout the enterprise. This heterogeneity has created islands of information as different types of systems have been introduced. Businesses rushed

in new systems and new technology without putting enough foresight into future architectural needs. [2] Indeed, developers of information systems were rewarded for being on time and budget, rather than making systems able to integrate with existing applications. [3] In addition to the assorted collection of systems, IT-decisions were made at the department level with solutions and technologies introduced locally eventually resulting in different applications in each department with duplicates over the business in large. [2]

Indifferently, the types of applications creating this heterogeneity all have their rationale and will continue to play a role. First are the packaged applications [1], i.e. standardised software packages not tailor made for the business need; that are increasingly bought by businesses. [2] Often one vendor is selected for one financial systems and yet another for customer relations systems. Systems like this, when not customised for integration, is hard and costly to integrate. One figure say that 35% of the development time is spent on integrating systems. [2] Second are that legacy-applications have become much entrenched in the businesses that they are undesirable to replace. [1] Third are business tailored applications that inestimable to meet specific organisation requirements. Last are database management systems that support the different applications. Since all applications do not work with all databases businesses often need to employ databases of various vendors and versions. [1]

Businesses therefore need to work with integration to gain a competitive edge by achieving the benefits as well as increasing the efficiency in their IT infrastructures. In addition there is a need to completely automate business processes and unify information systems with technology providing a flexible, manageable and maintainable IT infrastructure. [4]

2.3 Functionality

From a technical viewpoint, EAI has developed from traditional technology to more modern solutions. The traditional approach to integration was connecting systems in pair resulting in point-to-point integration. Each pair was glued together with technology- and application-dependent software called middleware. Middleware is a tight form of coupling where transport protocols and document formats need to be harmonized. Solely being a technical solution it usually did not include any awareness of business processes or related transactions taking place. [2] [3] [5]

Even though systems got integrated and low-level communication could be abstracted with traditional approaches there are multiple drawbacks. First, as two systems get tightly paired it becomes harder changing either of them without impacting the other. Second, with systems integrated in pairs the number of links multiplies when more systems are added, and ultimately results in an unsustainable and unmanageable situation. Systems become harder to maintain and additional integration become an arduous effort. Third, the technologies themselves are not very efficient or maintainable, and can be even risky with insufficient error handling. [6] One case study corroborates this by discovering that: implementation of new business requirements are long and time-consuming, there are escalating costs of maintaining interfaces, e.g. at the financial institute 40% of the budget was spent on maintenance; and finally changes in one component can unexpectedly propagate through the environment and by that make problems harder to isolate and changes more risky. [1]

Instead, recall from the definitions of EAI the purpose is not only to connect applications. Rather it is to take a more holistic approach and enable efficient and flexible information sharing for business processes, and ultimately managing the information instead of sharing it. This calls for more advanced middleware and new technology. As a substitute of point-to-point integration, EAI promotes software that can connect many systems to many (many-to-many). In such solutions systems are only integrated with the middleware itself over a few interfaces. The middleware are then responsible of routing messages between the different systems. [2] To illustrate the difference in complexity compare the number of links needed, with n system to integrate, in a one-to-one integration to a many-to-many middleware. In the former are $n*(n-1)$ links created compared to the n links in the latter. The difference is pictured in Figure 3.

By employing the principles of many-to many yet further possibilities can be added, for example workflows that can manage whole business processes and integrates them with systems and people

throughout the business. New business processes can thus be enabled as needed when the infrastructure already is in place. [3]

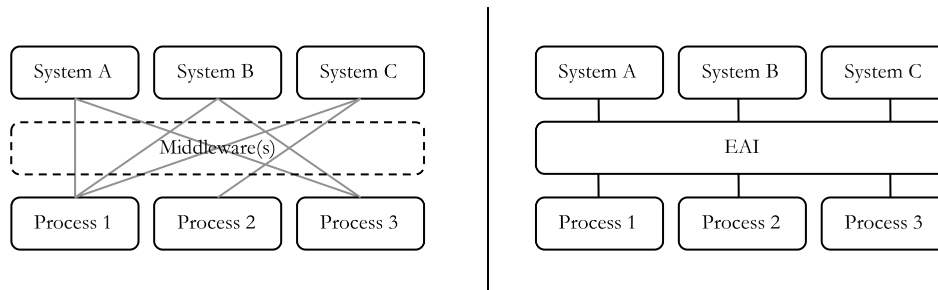

Figure 1 - Point-to-point integration (left) vs. many-to-many integration (right)

Layers and Maturity

Methods for EAI can be complex and varied with many different methods and techniques. [2] Instead of discussing each individually, EAI functionality can be divided into four different layers. From the bottom up are first the communication services, or data transport. This layer is traditionally based on remote procedure calls but now normally based on message queuing as transportation of data. Second are distribution services, or message routing, that sends the message to the right application. Routing can be either point-to-point or publish/subscribe (many-to-many). Third are transformation services that conforms data to the application input format. It might also validate and consolidate data, the latter for example if one application needs input from multiple sources. Last, and at the top, are process management that includes business processes and rules. This layer coordinates the transformations layer to follow business processes and rules applying to it. [2]

What functionality that is present depends on the business' implementation level. Such implementation levels or maturity levels is defined by Schmidt [7] in five levels:

- Pre-integration: stand-alone systems with manual data synchronisation. Processes are separated and are little reused.
- Level 1: Point-to-point integration is present through message-orientation and loosely coupled systems.
- Level 2: Structure integration by a hub, star or bus. Middleware are present with message brokers, transactions, rules processing and data transformation.
- Level 3: Process integration is achieved by not merely sharing information instead managing it between applications, i.e. the process is steering the information instead of being steered. This involves process automation tools in form of automated decisions, workflows and routing.
- Level 4: External integration is in place. There are also more advanced middleware features as security, intelligent agents, data mapping, application semantics and interface adaptations; and common data standard across the application.

These four maturity levels can be correlated to the four functional layers presented. At maturity level 1, point-to-point integration is in place by using layer one and two functionality. Second level maturity add architecture to the solution and the use of message brokers, this allow for many-to-many (publish/subscribe), and involves functionality from layer one to three. Third maturity level adds further enhancements by introducing process automation and workflows that use functionality from all four functional layers. As with the third level, the forth use all functional layers, but to a higher standard with features like security, mapping, intelligence, semantics and interface adaptation.

Service orientation

A concept that originated from many-to-many middleware is the enterprise service busses (ESB) that have evolved in the field of service oriented computing (SOC). It is similar to EAI with a vision to put together applications "into a loosely coupled network of services that can create dynamic business processes and agile applications that span organisations and computing platforms" [8]. The difference is that SOC exclusively philosophise about applications as services, i.e. each application provide some set of functionality to the enterprise. In contrary EAI does not make such distinction and bridge data either means available.

The objectives of ESB are to loosely couple systems and break logic into distinct easily managed services [8]. Furthermore it uses middleware products found in EAI technologies together with orchestration and choreography technologies, i.e. the way technologies work together to form business processes. Altogether it provides a backbone for service-oriented architecture and makes it, similar to EAI, possible to create assemblies of applications and services to form business processes, which can automate business functions in a business.

Component-based Architecture

Similar and pre-dating the SOC concept of interconnected services is component-based architecture. With component-based architecture, much like in SOC, applications can be flexible assembled by services offered by different components (see Figure 2). It is not however an integration technique, instead it focuses on concepts and methods of dealing with information modelling. Trying to avoid expensive point-to-point interfaces the aim is instead to provide one reusable interface usable to many applications. [9]

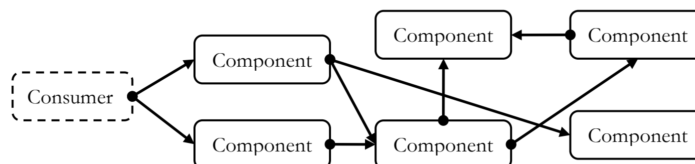

Figure 2 - Conceptual diagram showing component-based software (arrows mark dependencies)

2.4 Critical Success Factors

Critical success factors (CSF) aim to catalogue factors that are the essential for business success. It is factors that are finite in amount and defined for a narrow problem, in which if they all are satisfactory, will guarantee successful competitive behaviour [3]. Such factors are therefore a good foundation to base investigation on.

Prior research has strongly focused on technical solutions: few have dealt with application integration success. Critical success factors for Enterprise Application Integration is thus a young research topic with most of the prior work done in the last eight years. The works of the most relevant main contributors for areas covered in this report are outlined in this chapter:

Themistocleous 2004 – Introduces a model for EAI adoption by extending prior research on electronic data interchange (EDI), an electronic way of exchanging data between organisations. Ten influential factors for EAI adoption are identified: costs, barriers, benefits, internal pressures, external pressures, IT infrastructure, IT sophistication, support, the existence of a framework for the evaluation of integration technologies, and a framework for assessment of EAI packages. [4]

Yet only some are completely or partially applicable as success factors as the model also includes driving forces and motivations for EAI. Thus, aside from the factors: costs, benefits, internal pressures, and external pressures; the following success factors can be found:

- The need for employees with EAI skill as general information systems projects differs in various ways from EAI.
- Dedicated integration infrastructure to have a robust environment for integration.
- Evaluation framework for the selection of EAI tools. This to guide in the “extremely complex” marketplace of EAI products.
- IT-sophistication: the level of understanding and addressing of technical issues at an enterprise level.

Lam 2005 – Critical success factors based on literature are investigated and a case study is performed to validate these and output a more structured and holistic CSF model. Three broad groups of factors are identified, namely: top management support, overall integration strategy, and EAI project planning and execution. Groups and their primary factors are displayed in Figure 3. [1]

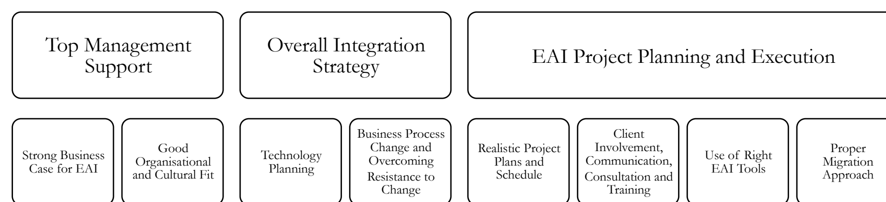

Figure 3 - Lam's CSF model

Many of these factors are overlapping with factors from traditional information system projects. However there are some unique and distinctive factors to EAI. The selection of the right EAI tool is one, and not surprisingly, unique to EAI project. Technical planning is another distinctive factor motivated by the need of upfront planning to understand what application to be integrated, how they communicate and create an integration plan.

From this research some additional implications can be drawn:

- That the skill set needed for successful EAI project is different from general information system development.
- Business integration (analyses and improves business processes) and enterprise data model (what information, and business rules applying to it, is distributed across applications) should be in-place before integration can begin.
- Organisations should consider availability of adapters as one of the main criteria when evaluating EAI tools.
- Adapters for custom or less-common packed-applications are usually not available resulting in custom integration. This thus affects the choice of EAI tool as some have better support for custom adapters than other.
- Use of a phased rollout strategy to make cutover more manageable.

Mendoza et al. 2006 – Introduces a framework of critical success factors according to organisations current integration maturity [7] and other infrastructure characteristics. The researchers are proposing 20 CSFs (see Table 1) at Schmidt’s four different maturity levels.

Table 1 – CSFs and what maturity levels they are related to **[3]**

CSF	1	2	3	4
Appropriate configuration of the communication software	●			
Standard data model documentation, unification and updating		●		

Appropriate outsourcing management		●		
Known organizational structure			●	
Change determined and justified at a productivity level				●
Valuable support by senior management				●
Appropriate strategy of security				●
Effective outgoing and incoming communication				●
Significant administrative support for the project	●	●	●	●
Complete technological infrastructure	●	●	●	●
Effective project leadership	●	●	●	●
Valuable project management	●	●	●	●
Relevant user involvement		●	●	●
Effective internal and external training plan		●	●	●
Effective organizational change management			●	●
Low impact of information systems on the organization			●	●
Careful strategy of implementation			●	●
High-expertise project team			●	●
Helpful technical support			●	●

Gericke et al. 2010 – are identifying (mainly from sources mentioned above) twenty-six success factor candidates from which, through measuring their effect on five success indicators, they devise twenty-seven hypothesis about success factors for application integration. From these hypotheses four groups of success factor are identified of having a broader impact on application integration, namely: Architecture Management, Organizational Maturity, Consolidation of Applications, and technical infrastructure. Other yet influential groups are: Business IT Alignment, Use of Methods, and Service-Oriented Architecture.

The five success indicators used were the achievement of: target quality for business process support, user satisfaction, time-to-market by the timely introduction of new products and its key to business success, architecture flexibility by that successful integration should lead to an agile architecture, and application integration cost goals by measuring the net benefits. From the empirical study hypothesis were drawn based on how a critical success candidate affected the success indicator (e.g. steering architecture with scorecards leads to shorter time-to-market). Being twenty-seven hypotheses, the relevant for this study are presented in the next section.

2.5 Reference Framework

When developing new applications some of the factors discovered in literature are more applicable than others. This section will select and motivate which of the previously discovered factors that can be applicable for development of new applications, in contrary to business-wide EAI rollouts as in some of the studied literature.

Among the success factors in current EAI research, ten have been selected to have more effect than others on development of integrated applications. First, without any order, is the use of *scorecards* to control architecture with architecture metrics [10]. Integration can be controlled and managed in the specifications and implementation phases to be successful. As developers get guidance in there work it leads to a shorter time-to-market. Another success factor providing developer guidance is the use of *principles and guidelines* to enable systematic development of to-be architecture for new applications. [10] Instead of following believes about a to-be architecture, developers can be guided from start towards the aimed architecture. This in turn leads to better reaching costs goals [10].

Important are also the presence of *integration architecture* throughout the business including all departments. To increase productivity and lessen the time spent on integration it is vital to understand the relationship between the various systems and how they connect. Having an overall architecture is essential for everything to have its place. [1] The use of scorecard, principles and guidelines shows the importance of communicating the architecture.

Further helping developers are *well-specified interfaces* as it is a prerequisite of understanding and investigating how to integrate with a system [10]. Interfaces that are well specified can be implemented more quickly when new applications are leveraging other systems. Well-specified interfaces lead to shorter time-to-market and reaching cost goals. Depending on the accuracy of interface specifications *feasibility studies* can lessen the number of unexpected errors and issues. [1] Few specifications are 100% covering and use feasibility studies can detect possible problems early on in the development processes and problems downstream can be avoided.

The infrastructure itself are also important when developing applications in an integrated environment and hence the success factor of *integration infrastructure*. [3] [10] [4] An infrastructure needs to be ready with integration tools and project development. When developing new application a well working infrastructure is indeed important.

A *common data standard* is also an essential success factor [1] [10]. High level of standards usage leads to shorter time-to-market and the reaching cost goals [10]. The availability of common standards means familiarity across the business and less reliance on personnel with special skill. This eases new application development when developers already are familiar with the data standard and able to work with it straight away. Similar to a common data standard are *few integration tools* [10] that also ease implementation by allowing developers to focus on a few techniques and tools. By keeping down the number of tools, so is also complexity. A small number of integration tools lead to shorter time-to-market and better reaching of cost goals.

Key for developers to fully grasp the business and its data is an *enterprise data model* [1] [3]. Projects may be completed more swiftly if an enterprise view of the data exists [1]. The enterprise data model shall also be unified and updated [3]. Having an enterprise data model makes the domain easier to understand for developers with the same concepts are presents throughout the business.

Adapters are another success factor and have a central role in EAI by being the bridge between the middleware and the application [1]. There are two categorisations of adapters, pre-built adapters and custom adapter development. Pre-built adapters help ease integration by providing working plumbing that results in less time spent on integration and more on business logic (and probably even less time spent in total). The ease of developing custom adapter is obviously directly improving productivity and decreases time spent on integration.

Last but imperative are the people that are working with integration and the importance of *employees with EAI skill* [1] [4]. Businesses need to inventory their EAI skill set and make sure the competences necessary all are involved. This is crucial even with an extensive set of architecture, and principles and guidelines. Special cases and rare problems will need to be addressed and it is therefore important for a business to have the right knowledge and to have it available.

The above selected critical success factors and their sources are summarised in Table 2.

Table 2 – Summary of applicable factors to new software development

Factors	Source	Consolidated factor
Steering architecture with scorecards leads to shorter time-to-market	Gericke et al.	Scorecards
Principles and guidelines better reaching costs goals	Gericke et al.	Principles and guidelines
Well-specified interfaces leads to shorter time-to-market, reaching costs goals	Gericke et al.	Well-specified interfaces
Dedicated integration infrastructure leads to shorter time-to-market and better reaching costs goals	Gericke et al.	Integration infrastructure

Complete technological infrastructure	Mendoza et al.	
Dedicated integration infrastructure	Themistocleous	
High level of standards usage leads to shorter time-to-market and better reaching costs goals	Gericke et al.	Common data standards
Common data standards	Lam	
Small number of integration tools leads to shorter time-to-market and better reaching costs goals, and target quality	Gericke et al.	Few integration tools
Integration architecture	Lam	Integration architecture
Enterprise data model	Lam	Enterprise data model
Standard data model documentation, unification and updating	Mendoza et al.	
Availability of pre-built adapters	Lam	Adapters
Custom adapter development	Lam	
Feasibility studies	Lam	Feasibility studies
Employees with EAI skill	Themistocleous	Employees with EAI skill

3 METHOD

This thesis report is designed as a descriptive case study. Each phase will therefore follow this approach as the case study research strategy spans from research design, through data collection to analysis. The method is suitable because of the strong contemporary focus, in contrast to studying the inert past in which a historic strategy would be more fitting. It is also fit since the purpose of this study is more open than 'what', 'how many', or 'how much' type of question, which would benefit from other strategies. Furthermore a case study can easily be based on multiple types of source, from documents to observations, both quantitative and qualitative. Yet case studies do not always need to have hard sources of evidence, for example is some observations hard to document. [11]

There are five rationales for single-case studies. Two of them are applicable in this case. First, since this is a *unique* possibility for the author to study the subject from within a large company. Second, since the case is *representative* or *typical*, i.e. findings may be valid for other organisations as well. [11]

Furthermore this study follows the five components of case study research design: a study question, proposition or purpose, a unit of analysis, logical linking between data and proposition, and criteria for interpreting findings [11]. The work therefore followed three major phases. First a literature study aimed at uncovering relevant aspects of application integration and build proposition, second collecting of empirical data, and finally analysis combining the empirical data with literature. The following sections will describe the different phases one by one.

3.1 Literature Study

The literature study began by investigating Enterprise Application Integration from critical success factors developed in the field. That gave an overview of all different aspects of EAI and allowed for further research into more specific parts and technologies. Also required from the literature study was basic understanding of the technical aspects of the field. But also what literature tells about the architecture deployed at the studied company and how that eventually can be connected with EAI.

A product of the literature study is the reference framework, or proposition, that has been worked out from the critical success factors found in literature. As there were multiple sources adding factors, duplicates and similar ones were consolidated. Factors that were not directly applicable when developing new applications, but rather aimed at company-wide deployment of EAI were not taken into account. Since no prior work for categorising factors existed, a naive intuitive approach was used during the selection.

3.2 Collecting The Evidence

The empirical data for the case study is gathered by conducting a project from within large transaction processor and IT provider in the travel industry. The choice of environment to study is motivated by:

- The sheer size of its business and its similarities with other companies, and
- As a large IT-provider it has a variety of different systems and products that, if not already present, require EAI, and finally
- It is similar to the study performed by Lam [1] at a financial institute, making it comparable.

Furthermore there are six major sources of evidence in a case study: documentation, archival records, interviews, direction-observations, participant-observations, and physical artefacts. [11] Appropriate for this study are participant-observations, documentation and interviews; all explained below.

Observations

Observations in common are strong when covering events as they occur and to get the surrounding context in which they happened. Observations can be both formal, with observational protocol, and casual using experience from the observation as evidence. Participant-observation specifically can provide useful insights into questions not asked or behaviour not observable from direct observations, e.g. the insight needs to be experienced. One very fitting example is how a piece of technology actually is use and what problems surround it. [11] Without observations researchers are stuck with ideas how it should work or must rely on potentially biased user stories.

Yet there are some drawbacks of using observations. For instance the possibility of the observation affects the outcome of events. [11] In this case however the observation impact on many aspects of the middleware is limited by it being out-of-bounds for manipulation. There is of course a possibility that the observation itself is triggering or causing problems. These can however be sidestep by an unbiased and observant investigator. The ultimate risk here is probably that time is spent investigating matters caused by the observation, than wide spread misconceptions that results in major flaws in the conclusion.

Another possible drawback is the limited coverage that an observation can be restricted to. [11] Maybe the research does not uncover all interesting applicable facts. Regardless, that problem exists in almost all research based on the simple principle that you can almost never be sure to have covered everything. One final drawback is that observations can be time-consuming as time is spent solely making observations. [11] However such thorough investigation combined with participating is hardly a waste of time. Although drawbacks observations, both direct and participatory, it is a valid source of evidence given the knowledge of these dangers.

Interviews

Interviews can be both formal by following a questionnaire and casual. [11] By that participatory-observation being the core of this case study, the most common and natural type of interview will be casual conversations that spontaneously occur each day. Because of the purpose of investigating, rather than proving or testing a hypothesis, no formal interviews with pre-decided questionnaires were hence used.

As with observations, interviews have some weaknesses. It can be biased due to the construction of questions, and the answers might be biased [11]. The first does however mostly only apply to formal interviews, which in this case will not be used. The latter however need to be taken into account even with casual interviews as people generally have different opinions how well for example a system works, or if it is easy or not to work with a certain product.

Interviews are also subject to reflexivity – interviewees answers what the interviewer wants to hear. [11] This is similar to that observations may affect the outcome of events. In a casual context, critical thinking and multiple sources can avoid that the research is controlled by for example by one employee. On the same note it is important to frequently take notes of casual interviews as one a further threat to interviews are bad recall of what was actually said. [11]

During the course of the case study many different people will be casually interviewed either in regards to problems with the participatory work, or information learned during meetings. When support was needed with the work, instant messaging or face-to-face chats were used. The roles that were interviewed are the following: product development software engineers and functional analytics, team-leaders, core integration software engineers and department managers. Since logging was turned off in the messaging client and many conversations were held in person, it was troublesome to keep a chain of evidence. Instead one needs to rely on memory and to take notes after a helpful interview.

Regarding the possibility of bias, since it was already hard to find the right people it was even harder to verify every account from different sources. Instead the report will be sent to key figures for validation.

Documentation

The last source for evidence to be used in this case study is documentations that will include meeting minutes and existing software documentation. Documentation is a good source as it is stable, i.e. can be reviewed again and again; unobtrusive, exact, and usually have a broad coverage. Documentation is useful in case studies for corroborating findings from other sources and to make sure concepts, titles and people's names are correctly spelled. Altogether it is certain that documentation play a clear part in any case study. [11]

However, documentation might sometimes be hard to retrieve, and all needed information may not be available. Also, as with most sources of evidence the use of documentation cannot be sure to be unbiased. Either by that the author being biased when writing the documentation, or that the documentation was selected in a way to reflect one truth rather than another. [11] It is therefore important to also view documentation critically and why it was given.

3.3 Analysis

There are some general strategies and some specific methods useful when analysing data. A general strategy provides guidance on how to treat evidence and produce analytic conclusions. Additionally a specific method can be used within any of the general strategies and provides even more detailed ideas on how to analyse data. Six examples of specific methods are: pattern-matching, explanation building, time-series analysis, logic models and cross-case synthesis. [11]

This study will rely on theoretical propositions as a general strategy, i.e. follow the propositions that have been developed during literature study to guide the analysis. In addition to the general method pattern-matching will be used. In large it concurs with the principles of the general strategy by a theoretically developed pattern being compared with an empirical based one. If they coincide it strengthens the proposed case. For example the presumption that complete technological infrastructure is important when integrating new applications. Observations of different infrastructures and their effect on integration can then be matched with the pattern and the theory can by that be validated or discarded.

Research into pattern-matching suggests using rival explanations, i.e. not agreeing with the main thesis, when there are independent variables. However as the prior research into critical success factors for EAI is rather inadequate, few such explanations are developed. Instead the pattern-matching methodology will be applied on the theories that are present in research today and matched to findings in the case study. From that discussions will be held on explanations for whether matching or not.

4 EMPIRICAL DATA

This section will describe the case study project itself, the organisation and environment it was conducted in, and finally the major integration phases of functional analysis and implementation.

4.1 Case Details

The case study was conducted at a large worldwide transaction processor. It employs a centralised communication middleware based on structured messages. The goal of the studied project was to develop a prototype that leverages existing systems and their functionality. During the project work was divided into three major types of phases. First there was a pre-study deciding the direction of the project, then functional analysis and implementation.

Strictly speaking the functional analysis phase and the implementation phase are consecutive of each other. However in reality they can be done alternately starting with functional analysis for some part of project, with sufficient of information available implementation starts, and the analysis moves on.

Functional analysis is the activity where given the domain problem a suitable solution is found in terms of what existing functionality can be leverage or modified to satisfy the problem. Given knowledge about what services to be consumed and roughly how, the observed process generally involve studying the message, testing it to verify the expected behaviour, settling for a solution, and then moving on to implementation.

During implementation the tested integration with other systems are implemented in code and connection setup to make it work with the integration middleware. The outline of the project is displayed in Table 3.

Table 3 - Project Outline

Phase	Activities
Pre-study	• Research into the domain of prototype • Competition Analysis • Study of the integration environment • Study of existing functionality
Iteration 1	• Functional analysis • Application Architecture • Implementation of core integration components • Implementation of first user interface (out of scope) • Implementation of iteration 1 domain specific functionality
Iteration 2	• Functional analysis • Implementation
...	
Iteration 4	• Functional analysis • Implementation
Closing	• Presentation • Documentation

The participating observer had no prior experience of the environment but sufficient knowledge of the general programming tools and techniques used by the business. As the integration middleware is proprietary, even knowledge about similar products was hard to apply.

4.2 Organisation

The company was originally created as a neutral distribution system between different businesses. The business is divided between two main and related business areas. Distribution that provides, among others: searching, pricing, booking and ticketing. IT Solution that delivers technical business processes support to travel providers, many of which are hosted solutions. The business model is booking fee or transaction based and mostly business-to-business.

As a global player spanning many different business areas, it has as considerable array of applications and with big share of legacy applications. There are many interconnecting application with queries that span multiple departments and services.

Organisational the business is divided into three major parts: marketing, development and operations, with headquarters in different cities. Figure 4 shows an overview of the organisation. Department B is mainly responsible for core distribution products and also common tools and infrastructure, whilst other departments concentrate on a group of products i.e. development of IT systems supporting the businesses in different markets. At the department level are developers organised into two types of teams. Product definition that is responsible for specifications and testing, and developers that develop the code by specification.

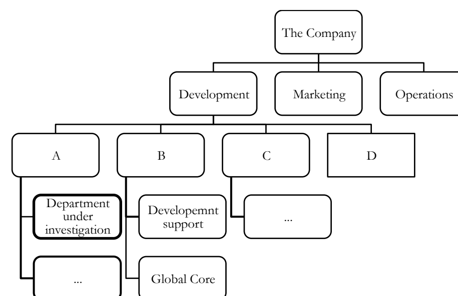

Figure 4 - High-level organisation chart.

4.3 Environment

IT-environment at the company is not that heterogeneous as one would picture from theory. Backend systems are written mostly with the Open Transaction Framework or are based on a legacy mainframe system TPF, a real-time operating system. Both are integrated in the service integrator and thus communicable in the same way.

Integration solution

Vital to the infrastructure and integration is the communication middleware that is the main integration point both internally and externally. The middleware provides load balancing and routing for point-to-point integration. Main design goals are performance, reliability, and service abstraction [12]. Clients and services are connected through TCP/IP sockets and messages are decoded and encoded in microseconds. Redundant components are used to avoid single point of failures (e.g. the norm is to always establish contact with the middleware over at least two end-points). Figure 5 displays a conceptual overview of the middleware. Note that the arrows in this case do not indicate integration between systems rather it is the connection network topology.

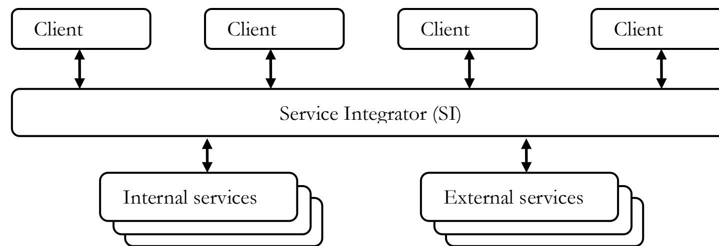

Figure 5 - Conceptual overview of the Service Integrator

The middleware is communicating through messages using EDIFACT, a United Nations defined electronic data interchange standard. EDIFACT is therefore the common data standard used throughout the organisation. Services and the middleware are closely coupled since functionality such as routing is directly present in the message format, and that each service is publicised by a corresponding message (a service and a message are therefore referring to the same functionality).

An example of an EDIFACT message is shown in Figure 6. Routing information and the message name is highlighted in bold (6XPPC is the sender, LHPPC is the target and the message name is PAORES). Each line separated by ‘ is a segment that can be shared across messages.

```
UNB+IATB:1+6XPPC+LHPPC+940101:0950+1'
UNH+1+PAORES:93:1:IA'
TVL+240493:1000::1220+FRA+JFK+DL+400+C'
PDI++C:3+Y::3+F::1'
APD+74C:0:::6++++++6X'
TVL+240493:1740::2030+JFK+MIA+DL+081+C'
PDI++C:4'
APD+EM2:0:1630::6+++++++DA'
UNT+13+1'
```

Figure 6 - Example of an EDIFACT message

Multiple environments with different configuration are also used during different stages of development and production. There is one production environment to which components gets promoted after extensive testing throughout the multiple test environments. During development there are mainly three relevant ones, first is DEV where there are little constraints, second is PDT where components are tested by product definition test scripts, and finally UAT where components are more stable but still in the test environment.

Using the functionality layers identified in 2.3 the middleware has the functionality of communication service layer and distribution layer, i.e. it handles communication and the routing of messages. However data transformations, business processes and business rules are not supported. Instead such functionality needs to be attended to in each component. Furthermore from a maturity viewpoint (also defined in 2.3) the middleware are employing functionality spread across different layers. Level 1 functionality is present by point-to-point integration, message oriented middleware, and the somewhat loosely coupled system. Interface architecture from Level 2 is also present. From Level 3 are none of the features present. Yet the middleware have characteristics of Level 4 by external integration and using a common data standard. There are therefore possible improvements by include process automation tools, workflow modelling, automated routing and decisions found Level 3.

Architecture

Architecture at the department is component-based. Business logic resides in back-end components that are exposed by services published in the SI. These services are then interconnected and together

form functionality. From this perspective, the architecture can clearly be said to be component-based. Conversely from a broader perspective the platform can also be viewed as a black box with a set of exposed services. The difference might seem small but the former would entail integrating with any component inside of the platform, possible with services not even engineered for that purpose. Instead the black-box concept make platforms expose only some public services and can take responsibility that functionality.

Back-end components are written using a framework called Open Transaction Framework (OTF), written in C++. The mainframe system TPF is gradually replaced by the component-based architecture.

4.4 Functional Analysis

In the functional analysis phase engineers work out how requirements can be satisfied from a functional perspective. What need to be changed, which system can or must be integrated? Answering these questions gives the analyst at an early stage an idea about the final solution. This phase thus consists of the following activities: studying, idea generation, and test of ideas.

Specification and Documentation

First are studying of specifications and other material for the systems or components that most likely to be used. What systems to study come from experience, new analysts therefore need to rely on more experienced ones pointing in the right direction. Specifications stand-alone are very strict and it is hard to get a good understanding of systems as a whole. Knowing whom to contact and understanding the organisation and the different responsibilities are thus important for efficient analysis. This is valid for new analyst, for more experienced ones where to look and what systems are interesting is less of an issue.

Specifications are available from different sources, some in a document database associated with the emailing platform, and some on intranet portals. There are also team-based wikis with some documentation or articles addressing different subjects. Employees new to this structure need direction to find the right document in the right place. Although there are some directions to where specifications can be found it is not always clear which is the correct current version. Use cases and descriptive texts are stored in a repository from where information can be retrieved. This centralised system is however not used as final specification store as could be efficient to do. Instead documents are generated from the use cases and texts to build a specification. By using this system as the final store for specifications they could more easily maintained, and browsable when use case can be directly linked to the next one, and the one after that, etc.

Ideas and Testing

Parallel to studying, ideas are generated on how to satisfy requirements. During this project at least three ideas surfaced. An idea can be an assumption of how the system can be used or other concepts on how the system could work together. The purpose on the idea generation is to formulate something that can be tested to see if it is a valid solution.

When testing ideas and assumptions made about functionality, services are directly invoked through the middleware and ideas therefore tried in an environment similar to production. For example, from studying the specifications one hypothesis may have egressed that component A and B would work together in some way. This interaction is possible to test, and by that verify that the assumptions about the functionality were correct.

However to setup such test script more detailed knowledge is required on how to interact with the actual services. As specifications are functional they do not themselves describe how to integrate with the component. Often is only the service name and what use-case it refers to available. Instead information on how to interact with the service must be looked up from its message structure. Message definitions are kept in a repository and are available from different tools. One tool for this is

the EDIFACT Viewer, a web-based view showing the message structure. Another is an editor called Edifact Editor that provides the user with the message structure and the possibility to get the resulting plain-text message. The latter tool is essential due to the hierarchal structure of EDIFACT that make messages prone to errors if craft by hand.

The generated message can then be tried with another important tool called Test Tool Server that is capable of injecting messages in into the message middleware. Pattern matching is used to verify the output. However when trying messages it is clear that definitions do not always tell the naked truth. For example fields marked as conditional are sometimes needed anyway, and some messages depend on prior messages being sent. This is not evident from looking solely at the specification or message structure. Instead are regression test scripts in practice the best source to lookup a message usage. This however is an awkward source of information as it neither entails all the possible usages nor in which context it should be used. Developers thus risk ending up with examples of a message working in one specific case but not in other scenarios.

Errors

During this phase of testing and experimenting it is common to get errors. Errors can be of different types depending on their source. The most common error type was the CONTRL error triggered by the middleware itself. This could be due to many reasons and the only indicator pointing towards the cause is an error code. The error code can be somewhat helpful if explained by documentation. This documentation was however hard to find and several co-workers had no idea of its existence. Although documentation became available there were still some errors hard to decipher. For example messages are sent to the wrong SAP the middleware responded like the service was there but without any possibility to make sense of the error. The other types of errors are sent from the components it and are more descriptive in their error messages as a result of the more detailed knowledge about the service.

4.5 Implementation

After functional analysis finishes the implementation phase can begin. Depending on the project model and development method used, the implementation of the application might have started well before functional analysis is done. The implementations of a service however, usually and in this case, does not start before functional analysis for that service is completed and scenario for a final idea exists.

Platform and Architecture

The project was developed using the Java development platform. It includes use of the: Java programming language edition 6, Eclipse integrated development environment, Wicket web framework, Maven dependency handling and build management, and other proprietary development tools.

Given that the company employs a component-based architecture, applications are usually developed in two or more distinct components, for instance one back-end component containing the business logic, and one being the front-end web application. However as the project was limited in resources, and with the objectives of a prototype, it was decided to develop it merely as a front-end application. As a result the application logic and front-end came to reside within the same component. The project could thus focus on business aspects instead of the intricate workings of back-end components. Similarly this case study could focus on one technology for integration rather than one for the front-end components and yet another one for back-end components.

The internal architecture of the applications was developed with a clear distinction between state and behaviour. Following a practice called Domain Driven Design (DDD). Domain objects represent state, services handle behaviour (i.e. modifying state), and the state is persisted using repository objects that integrate with the service integrator. This results in horizontal layers with separation of concerns and abstractions between layers when each layer has different responsibilities. The user

interface is responsible for communicating with the user, i.e. displaying and collecting data and passing it to the service layer. The service layer applies business rules and transformations to the data and saves the changes to the repository. The repository in turn calls executors that are responsible for interacting with the service integrator and "executing" messages. Interactions between layers are displayed in Figure 7 and a full class diagram can be found in Appendix A.

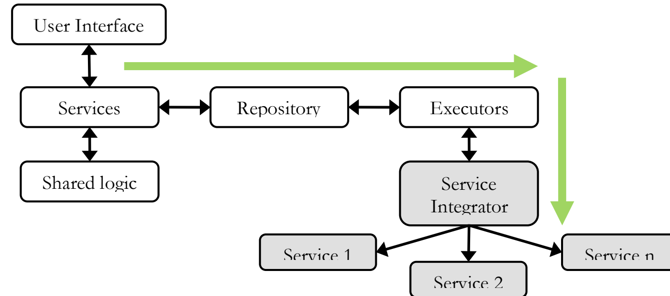

Figure 7 – Information flow

With this application architecture, and even with business logic and front-end combined, the final prototype still employs a serviced-based architecture with business logic structured as different consumable service. It would therefore be fairly easy to in the future publish the plain business aspects of the application on the service integrator through these services, even though the application not being back-end component.

The final solution integrated with four systems using thirteen services. A typical interaction is displayed in Figure 8.

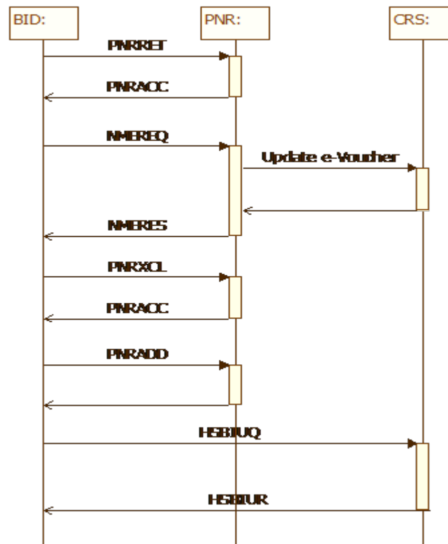

Figure 8 - Example of interaction described with UML

Implementing Services

Once the messages have been verified and tried, they are implemented and called through a communication library. The library consists of two parts, one pre-built module and per-implementation generated classes that represent the message structure. When sending a message the message representation is filled with data from the client code and then transformed to the

corresponding EDIFACT message. The message is then sent to the integration middleware who responds and the process is reversed, i.e. plain text message to object representation.

In addition to correctly composing messages they need to be sent to the correct target. Not all services are published on all integration points (SAP) and developers must therefore be careful to target the right SAP. A web-based tool called SI-Viewer must be used to lookup the service name and where it is published. In the production environment however there is usually one SAP per product, making this problem obsolete in that environment. Yet the differences between development and production complicate the code slightly and make it harder to maintain with two separate configurations. Mostly because of the time-penalty involved in creating new SAP:s.

As described earlier the application is built in different layers. The interesting layer from an integration viewpoint is the repository layer that consists of the repository classes themselves, executors and the JAPI library. The repository classes are the interface and logic that the rest of the application is communicating with to retrieve and save state. For example, if there is a class named auction, there are probably a corresponding repository named auction repository that is responsible for the saving and retrieving of auctions. From the outside a repository is perceived like a list with functions as get one, get all, update, create etc. This general concept of a repository makes it a nice abstraction for the application layer, as it does not need to care about complicated specifics about persistence. See Figure 9 for picture describing the composition.

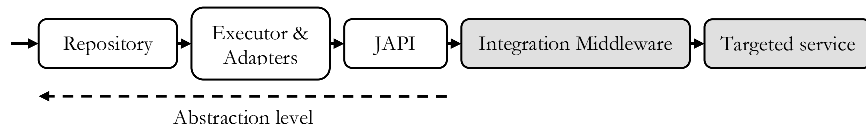

Figure 9 - Integration Components

Repositories in turn use an adapter (example in Figure 10) to translate the input from domain specific classes to the generated message structure representation. Once in that representation the JAPI library is called for the actual interaction with the integration middleware. After the message is sent the library waits for a response and returns it in the message structure. Control is then handed back to the repository that uses another adapter to transform the response to something understandable by the domain, e.g. a Boolean, an integer, or an auction object.

```
public static PnrRetrievalMessage newPnrRet(LocatorIdentifier pnr) {
  RetrievalMessage request = new RetrievalMessage();
  RetrievalMessage_facts ret = request.getInitRetrievalFacts();

  ret.getInitRetrieve().setType("2");
  ret.getInitReservationOrProfileIdentifier()
     .getInitReservation(0)
     .setControlNumber(pnr.getControlNumber());

  return request;
}
```

Figure 10 - Example of adapter code

Service outages

During implementation, and somewhat during functional analysis, calls to services occasionally resulted in erroneous responses. Services either timed-out or errors occurred on functionality that had worked earlier. There were no particular components that had problem, rather it were different components throughout the business. Time-outs were usually caused by failing back-end components that stopped responding because of various reasons, usually problems resolved themselves ones the back-ends were online again. Erroneous functionality however caused by change in functionality was

more time-consuming as it might take days for the service to be online again. Not all problems were triggered by change of functionality instead flaws got introduced by updates in unrelated project.

The reason for the dynamic test environment and the experienced problems is that it is actively being developed and new versions of components are daily release to it. Sometimes these are not well enough developed and errors therefore occur. Yet errors might also be due to faulty use of the service. One might have thought the component worked in a certain way one day by sending messages to it and verifying the output. But if that functionality is not in the specification or test scripts, it might be changed when the component is updated.

Services can be affected for hours up to days and developers consuming the service are usually unable to do anything but wait until the problem is resolved. Consequently valuable time is wasted and projects might be delayed if developers are unable to continue to work on different parts of the application. When errors occurred during the case study, the architecture of the application allowed the developer to continue to work on different parts as interfacing classes could swapped to ones simulating the response.

Table 4 - Overview of tools used

Tool	Description
Edifact Viewer	Web based browser for EDIFACT message.
Edifact Editor	Graphic tool for investigating message and generating plain text representation filled with values.
Test Tool Server	Tool to send and receive messages over the SI.
Visual Edifact	Tool for organising services and packages of them.
SI-viewer	Web interface for viewing the current SI configuration, e.g. where are services published and so on.
JAPI	Library for communicating with SI from Java-based systems.

5 ANALYSIS AND DISCUSSION

This section discusses application integration from the perspective suggested by the critical success factors extracted in section 2.5, with the empirical findings in section 4. Each headline addresses one or more critical success factor and the posing problems or satisfactory implementation of them.

5.1 Scorecards, Principles and Guidelines

From the result gathered during the case study the absence of tools to guide development of new application became clear. In the reference framework are both principles and guidelines and scorecards recommended and are expected to respectively lead to better reaching cost goals, and shorter time-to-market. Yet the business in the case study was relaying on functional specifications and a few key figures to help with integration matters. Such reliance makes businesses vulnerable of delays when the few key people need to assist throughout the organisations while focusing on their prime work task. As a result businesses will have longer time-to-market as opposed to shorter time-to-market achievable with scorecards. Solutions may also deviate with neither scorecards nor principles and guidelines available to enforce the integration strategy. Each team (or developer) might risk follow their of approach to integration, making it harder for later maintenance when each solution is different. Ultimately this will lead to increased maintenance costs and yet again further reliance on people instead of documentation.

Recommended are therefore to start use scorecards to control integration, and the use of guides and principles on how to design adapters and strategies for communicating with the integration middleware. But also guides on how to interact with the most common services. For example an updated developer's guide containing common scenarios and patterns could significantly help developers integrate with services and off load key competences. These components where all missing in the case study and even though it is not possible at this point to conclude how much impact such tools would have in regards to other factors. They are still important as if they were to be completely discarded it would even further add to the already highly heterogeneous environments with rashly new solutions, and development delayed by awaiting external help.

Adding these components does not have to be cumbersome. In the case of developer guidance it can be as easy as creating a short readme for each integration technology exemplifying common interactions and caveats, and in turn spend less time on specifications that are to massive too comprehend. Scorecards can however be more of a challenge, as it requires a continuous process of extracting, evaluating and changing the indicators to always be aligned with the strategy. Examples of indicators might be the number of integration technologies used, the number of integrated service, or the amount of money spent on the integration part of a project. These indicators can both steer as well monitor solutions, steering by setting goals for the number of integration techniques and monitor by having threshold for spending.

5.2 Well-specified Interfaces

Related to developer guidance is the need for well-specified interfaces. The specifications examined during the case study contained no interface specifications. Instead as each service was exposed by a message its structure served as the specification. Using the structure however poses three problems. First it does not reveal anything about the context. For example does a service require prior messages or some context to be set? Second, since messages are built from standardised segments (e.g. one segment representing a booking) conditional fields were sometimes found to be the opposite. There were numerous examples of messages being sent according to definitions with conditional fields omitted that resulted in different errors. Third, message definitions do not provide developers with possible errors and likely causes. Even with a very accurate interface description it is important for developers to be able to efficiently troubleshoot problems. In the case study most errors consisted only of an error code without any description. Together with the missing documentation such error

codes where found to be of little use and developers were yet again forced to depend on others for help.

To cope with some of the problems of using the message structure as interface specifications, regression test-scripts where often referred to as examples of valid interactions. Finding the applicable examples was however hard, but when found and together with access to the service-developer most problems could be solved. On the other hand without access using the examples alone, integration was often hard and time consuming as developers fumbled in the dark for solutions. Well-specified interfaces would therefore, as suggested by the reference framework, lead to shorter time-to-market when less time is spent on investigation and debugging invocations of services.

Some services had additional documentation that if found addressed some of the aspects mentioned above. However as these stored were commonly stored in different systems they were hard to find and again developers were forced to spend lots of time on quests for answers. This problem is not unique to EAI; rather it is a general problem in the field of information system development. Yet findable documentation is of great importance when solutions spans many departments and involves many different people.

Interfaces should therefore be specified with not only accurate information of valid interaction, but also their use-cases and context. Each use-case should include the valid combinations of input parameters the expected output. The same documentation should also include possible error codes with descriptions. One example of an easy to use and in comparison very modest specification is the Java API documentation. For each method, in this comparable with a service, it describes in a standardised format: the input parameters, the return data, any possible errors that might occur and why. Such specifications that also are searchable and to be found in one place would definitely help to improve efficiency and the daily work. For businesses in general and especially for the case study business, much can be gained by having fewer bottlenecks as a result of more self-reliant developers.

5.3 Employees with EAI skill

Another critical success factor is employees with EAI skill. Skills in this case include awareness of concept crucial for EAI such as understanding the distributionness of communicating with different systems over the network and how that differs from normal information system development. Furthermore it includes knowledge of the procedures for integration in place in the business and the exact integration tools being employed.

However hard to measure in values it was clear that the case study business have some key employees distributed thru the business that are more skilled than others. As mentioned many times before these people are essential in backing the whole business by guiding other employees to the right resources and helping out with their problems. Essential these people are huge connectors of knowledge that are of great value but by that also a big liability. If one was to quit, or god forbid die in freak gasoline accident, the efficiency among the supported employees would plunge. Instead businesses need to relay on education and incentives to make more employees self-reliant and able to develop and troubleshoot integration by themselves or in their respective teams. Knowledge needs to be attained by processes, not people. It is therefore important to increase the "bus factor", the number of developers that need to be incapacitated to make the business completely halt; and make the business independent of few key employees.

Interestingly enough, education itself has not been found in any prior critical success factor research for EAI. Neither has it been in the focus of prior sources about the importance of skilled EAI employees. Yet from this discussion, together with the need for proper documentation, education seems unmistakeable as one of the most important factors for successful application integration. Both knowledge about specific techniques but also how integration is employed in the business.

5.4 Feasibility studies

From the reference framework feasibility studies is another important critical success factor that should result in fewer errors and unexpected behaviour during implementation. One interview depicts the importance: "It's important that we did a feasibility study because quite a few unexpected errors and issues cropped up." [1]. At the case study company feasibility studies have a clear role in the project lifecycle, and has indeed proved its importance. As a part of the functional analysis phase, dependent services are tried with example invocations to assert the functionality. This ensures that when it is time to start development, as many questions as possible have been resolved and potential problems already been identified. And indeed, during the case study working in iteration with feasibility in function analysis and implementation alternately lots of problems could be detected even before coding begun.

However, as this adds to quality, the feasibility studies like the ones at the case study business might not be as efficient as possible. What businesses lack in documentation need to be covered by testing and feasibility studies, and by having better documentation less time can be spent on studies. This would ultimately lead to shorter time to market as well as other efficiency gains. For businesses that already have good documentation in place, feasibility studies can instead put additional focus on smart business solutions rather than struggling with disobedient services.

Not doing feasibility studies would risk make the development process fragile by an increased number of showstoppers and ultimately result in expensive rollbacks or changes amid project. Yet again, too much of a waterfall approach to development is neither good, testing the right things and doing the right proof of concepts are key.

5.5 Common Data Standard and Enterprise Data Model

Another critical success factor is a common data standard, which helps businesses reach cost goals and achieve shorter time-to-market. With a common data standard, employees will be familiar with any project within the business and be instantaneously deployed to projects without re-schooling. Tools and processes can furthermore be refined throughout the company by not having different departments working in different ways.

At the case study company a common data standard was in place by the use of a single message standard called EDIFACT. By exclusively using one data format in the integration middleware, utmost all systems used the same data standard for communication. Built around it are a larger number of tools and processes explicitly designed for the message standard. For example the message-structure-repository that centralises all messages and their segments; another is the validation process through with change committees. This amounts together to a good foundation for a common data standard that spare the business of much of the possible overhead mentioned above.

Yet there are some potential problems. First, as new technologies emerges, most importantly SOAP, an XML-based interoperability markup language; the data standard has become more of a hybrid between EDIFACT and SOAP. SOAP messages are being encapsulated into EDIFACT messages and routed as such in the integration middleware. To still have a common data standard, businesses in similar situations need to ensure that with additional formats being implemented, that it is well thought through and that the integration middleware does not end up yet another packet switching protocol. Instead it must keep its single common data standard or be rebuilt to natively handle multiple standards and the transformations in between. Second, as messages are kept in a central repository changes risk being slow by bottlenecks in the review process. This would rather lead to an increased time-to-market than a shorter suggested by the reference framework. However change should not be too casual as this risks breaking software elsewhere and that less thought is put into interface design. The challenge is to find the right balance.

Related to the common data standard is the use of a common enterprise data model. An enterprise data model is to give a holistic view of the data in an enterprise. For example answer questions such as: what are the properties of a customer? And, how where is such data stored and access? From an

integration viewpoint at the case study business, it existed merely as the message structure or the common segments in it, i.e. some broader concepts where shared across multiple messages. This does not however constitute to an enterprise data model as no holistic view over the data is provided. Rather are bits and pieces described here and there, but not necessary in a coherent fashion. When message segments are duplicated or messages designed without reuse in mind, discrepancies may occur with data being formatted and represented differently across messages. One examples of this from the case study is that dates were represented both as 2001:11:25 and 011125. Another is the inexplicit naming of booking references, in some messages *control number* and in others *locator record*. Yet it does provide some of the characteristics of a data model since some common concepts are present.

The absence of an enterprise data model mostly affects communication, as it gets more cumbersome with multiple names or ambiguous descriptions of the same concepts. With large EAI projects spanning multiple departments, having some delivering components with misinterpreted concepts can lead to costly delays or worse. Indeed, compare the American subcontractor to NASA that delivered data of their engines in pound force instead of the believed newton meters. As a result the spacecraft descended into and burned up in the atmosphere of Mars. However, as this report focuses on the development of new applications and that most such errors can be caught in feasibility studies, the more relevant issue is developer efficiency (and by that time-to-market). Not having the same definitions lead to the need for transforming and processing data in ways that would not have been necessary with common concepts. Furthermore with inconsistent naming solutions takes longer to investigate and develop since it might not be clear that the control number and locator record corresponds to the same thing.

5.6 Integration Infrastructure and the Number of Integration Tools

With the single message-based integration middleware in place at the case study company, the number of integration tools was kept low. Counting the supporting tools (Table 4) and the middleware itself the number is six. What constitutes few integration tools is not obvious but comparing with the use of multiple middleware with their own set of supporting tools, the use of one can be considered low. Hence compared with businesses employing many different integration tools, businesses with only one should have shorter time-to-market, better of reaching costs goals and target quality.

The integration infrastructure itself, described in 4.3, does not support high-level EAI functionality as workflows, process models and business rules. This is probably the largest and most profitable change to be made to the integration middleware. Prior research shows that by using notifications with a publish/subscribe pattern instead of passing messages point-to-point, the cost of interface maintenance can be kept down. If services need to update data in additional services, rather than contacting them individually, it publishes a notification to the integration middleware to which other systems subscribe to and act upon, e.g. update their data, or send an email.

As an example, suppose there are a central booking system that keep information about all aspects of a trip and therefore need to receive information from both hotel reservation systems and airline reservation systems. Today this exchange might be done through point-to-point connections from the central booking system to the reservation systems. With a publish/subscribe model the central system could instead request this information by publishing one request message on the middleware. Services subscribing to this event then respond the central booking system either directly, or by letting the middleware intercept and merge the different responses before they are returned to the central booking system. The former requires that the data being communicated back is conformed to one receiving interface and thus in fact form yet another point-to-point integration. Instead as in the case with the latter, services can respond differently in a format making sense to them and other system and before sent to the requesting service transformed in the middleware. Yet another possible solution with publish/subscribe is to upon a hotel booking publishing this directly to the integration middleware from which all interested services can listen to this event.

Solutions like the ones above help to detangle and structure point-to-point and component-based systems by moving the complexity of what services to contact and the interface implementation from

the components themselves. Instead additional services can be integrated only by registering them in the integration middleware without needing to implement additional interfaces to any existing components. This helps development of new applications as more focus can be put on business logic and the exact interfaces making sense for it.

During the case study one problem became clear when targeted the test environment. Services were sometimes unavailable for hours and even days as components were: upgraded, changed, tested or experienced problems. In integrated environments applications under development need to depend on the systems it is integrating with. Outages of key services thus dramatically slowed down the development process. However as in the instance of the case study application the use of clear abstraction layers could minimize the impact of such events, as it was possible to switch to fake representations of the real services. There are no comfortable general solutions to problematic, as services need to be tested and deployed in live environments. Yet what can be done is to have more rigours processes aimed at ensuring fewer outages, and have different tiers of test environment. One usual setup is to have at least four environments. Starting from the most casual one usually named development, then test where systems should work flawless in development to be promoted to, after that stage that is as like the subsequent production environment as possible. Another more low-level approach is to systematically incorporate fake services directly into the client interface code like the one described from the case study. However this risks being hard to maintain as interface and data evolve.

To concluding, by employing few integration middleware, the number of tools can be kept low, which ultimately has positive impact on time-to-market, reaching cost goals and target quality. Businesses that like the studied company only employ basic EAI functionality can benefit from extending the integration middleware with more high-level features. For example a publish/subscribe pattern that would help detangle point-to-point connections and the following complexity. Other interesting features are data transformations, workflow and business process modelling.

5.7 Adapters

Central when using a commercial enterprise service bus are the availability of standard adapters and the ease of developing custom adapters. [1] In the case of new application development, the latter are of most importance since most tailor-made applications rather few times comes with cord to plugin. Instead adapters have to be developed to communicate with the integration middleware and by that the remaining services. Making adapter development reliable and as frictionless as possible leads to increased developer efficiency and yet again focused be put on providing business value. Depending on the integration infrastructure being used adapters can be generated, described with a domain specific language or written completely manually. In the interest of efficiency the two former are to prefer.

At the case study business a custom-made in-house developed integration middleware was in place and any standard adapters or standard tools for adapter generation can therefore at once be ruled out. Instead the business had created its own set of tools that backed integration and adapter development. From the message definition the tools helped generate client code that provides communication functionality with the message middleware and abstracting the message format with more descriptive interfaces. Such tools are very valuable as developers can focus on high-level integration rather than sending the correct zeros and ones over a wire. However the generated client code were sometimes very verbose with long names that made it easy to get lost in what actual segment being set. Instead there might be a lesson learned about abstracting away too much of the underlying representation. Since most engineers mastered the jargon of the raw message definitions and commonly referred to in feasibility studies, translating to descriptive names was rather perplexing than helpful. Yet even though there are room for improvements, the effort put on such enhancements would probably be better spent elsewhere as long as the minimum requirement of having abstracted away much of the communication and fault-handling.

5.8 Integration Architecture

Architecture is broad subjects with many aspects and viewpoints. In the grand field of Enterprise Architecture is Application Integration merely one way of composing software together among others. Yet when employing an EAI approach to system collaboration is the architecture on how systems and components are to be structured and communicate very important. There are many ways of achieving this and the critical success factor stresses the important of such strategy.

At the case study company, an architecture was used there systems were divided into a front-end and a back-end part. These were written in different technologies and were handled very differently with different performance demands. The prototype that was created during the case study was developed as one component rather than to the normal decomposition of multiple components. In this case diverging from architecture had both advantages and drawbacks. The advantage was the ability to show that the quite strict breakdown between front and back-end might not be necessary. Instead of using techniques that were for one person deemed to hard to understand during six months, more lightweight techniques was tried for both the front-end and back-end. This resulted in less time spent on development and with a correct design also performant enough. Yet for critical core components where both performance and availability are of most importance, the more rigorous back-ends might be used. For more transient applications, e.g. prototypes or satisfying special customer requirements, more lightweight techniques should be evaluated.

The instant drawback of this experiment of using only one component was the ability from a "front-end" application to expose services consumable from other components through the middleware. Instead interactions had to go through other channels, e.g. web services or custom communication solutions, undermining the common data standard.

If this architecture separating back- and front-ends is to remain, front-end development could be simplified. Today Java is commonly used in enterprise front-end application for its stable track record and predictability. However it is not known for allowing rapid development, and with no business logic residing in the front-end applications such platforms are likely to heavy for an efficient implementations (like using submarines to fish for sea-bass). Instead thinner front-end technologies could be used to provide a more agile approach and simplified development. Dynamic programming languages together with highly popular web development framework could make front-end development a blaze.

5.9 Summary

As evident from above, all critical success factors discovered in prior research have a clear role in developing new applications in an integrated environments. From factors that prior research has considered in the context of enterprise wide rollout of integration solutions, it has been possible to show their importance when developing new application. When evaluated and discussed, much focus have been on education of developers and knowledge management. Numerous examples show that when applied, most success factors have ended up in discussions around the importance of the developer itself, and how to make him or her as efficient as possible while making processes owning knowledge rather then single employees. All of the following critical success factors are ways of archiving this: the use of scorecards, principles and guidelines, well-specified interfaces, EAI skill, common data standards, enterprise data models, and finally the number of integration tools. When put in such context the related critical success factors are all important in integrating new applications.

Focus has furthermore been on the technologies itself. Further critical success factors can be linked to have a great impact when developing new applications. For instance by employing more sophisticated with EAI methods to further decouple systems and keep the number of interfaces down to a minimum. This can be achieved by techniques such as publish/subscribe or workflows that are found in the critical success factor of Integration Infrastructure. Here are also the technical aspects of the adapters linking applications to the integration middleware important. By having more abstractions the more independent will the final solution become of the exact implementations of the integration middleware. It will be that also be more tangible for the future. Last but not least have the Integration

Architecture a significant role in new application development. Not only on the principles that there should be one present, but also the right one. By employing a for the business suitable architecture have a clear impact on new development. Would the architecture to demand systems being written using techniques not suited for today's modern business, it would certainly have a negative effect. Instead businesses need to employ agile and fluid architectures that can rapidly respond to challenges.

Altogether there are many important considerations to be made for business leaders on how to steer and align IT and the business. When convinced that integration is the right way to go, the critical success factors presented here are a good guide for starting exploring initiatives and ultimately succeed with the integration.

6 CONCLUSIONS

Enterprise application integration will continue to be relevant as grand enterprises continue to employ countless of different systems. Even though cloud computing and the move to the web relocates systems from the enterprise itself. Businesses will still need to integrate process over different systems and acts as one entity, no matter where the systems are located.

This report has described the benefits and driving forces of enterprise application integration including the problems businesses are facing today. Drivers such as globalisation and the new way companies do business introduced by the Internet, to advantages where businesses profit from operational, managerial, technical and organisational benefits. Problems have included heterogeneous and overlapping systems throughout the enterprise that have created islands of information.

Guided by the critical success factors for EAI that are applicable to new application development, this case study has furthermore discussed the aspects of EAI from real-world observations and experiences. From empirical data and discussion it is evident that all the selected critical success factors are valid by having meaningful input and real word impact. Absence of some has been seen to cause problems whilst all being present have helped integration. There have been two trends of problems during the discussion, developer guidance and documentation, and functionality.

By missing *scorecards* and *principles and guidance*, pressure has been put on key figures to ease integration. Similarly, developer will continue to be dependent on others in the absence of *well-specified interfaces*. Each unfulfilled factor aggregates integration by making it more time consuming, and ultimately more inefficient. This puts more pressure on the number *skilled EAI employees* to avoid bottlenecks. Businesses should therefore focus on making developers self-reliant and not in the hands of a few key figures to drive development forward.

The second trend was functionality and that businesses should leverage more advanced functions of EAI. By sticking to point-to-point integration costs for maintaining interfaces are higher and slower time-to-market. Functionality such as many-to-many (publish/subscribe), workflows to model business processes and rules should instead be used. This came to light from the critical success factor of *integration infrastructure* but was not closer detailed from prior research. Instead it was evident from the literature study of functionality that the level of functions is important for how successful an EAI implementation is.

Three critical success factors that were present and helped in the daily work were the use of *feasibility studies*, *common data standard,* and *custom adapters*. The first helped by saving hours of working founding problems late (kill the monster wile its little), the second helped by allowing competence to be focused on one data format, the third by providing a working library to develop custom adapters.

There was also one critical success factor that was of less importance in this study and did neither pose a problem nor any solutions. The need for an *enterprise data model* could have helped understanding and to enforce the formatting and representation across systems. However it was not present and in this case study it was found that parts an enterprise data model were handled by other critical success factors.

Future research

During the course of the study several interesting and so far unchartered aspects of EAI have surfaced. Future research into EAI could thus investigate:

- How should interfaces be document and made available?
- How should applications be architected to integrate robustly?
- To what extent is education important to the success of EAI?
- Can EAI succeed with point-to-point integration? Should the current practice be updated to many-to-many type middleware?

There are also several interesting future research question for businesses similar to the one described here. Since developer efficiency was questioned with the absence of much of the guidance enabling it. A study on how developers spend their time would help identify and prioritise addressing of the problems. Finally and partially out of scope companies should question and division between front- and back-ends.

Limitations

One limitation is the use of a single case study and that empirical data being largely based on participatory observations. To generalise the study multiple case studies should be used with a more formal observation protocol or other measurement. Furthermore may the outcome of this case study been affected by the fact that the application was forced to use services in ways never intended. Hence some problems that arose might not arise ordinarily. Nevertheless this report highlights some of the important aspects of development of new applications in an enterprise application integration environment.

7 REFERENCES

[1] Wing Lam, "Investigating success factors in enterprise," no. 14, 2005.

[2] Naveen Erasala, David C. Yen, and T.M. Rajkumar, "Enterprise Application Integration in the electronic commerce world," *Computer Standards & Interfaces*, no. 25, 2003.

[3] Luis E. Mendoza, María Pérez, and Maria Grimán, "Critical Success Factors for Managing Systems Integration," *Systems Development*, 2006.

[4] M. Themistocleous, "Justifying the decisions for EAI implementations: a validated proposition of influential factors," *Journal of Enterprise Information Management*, vol. 17, no. 2, 2004.

[5] Mike P. Papazoglou, "Service oriented architectures: approaches, technologies and research issues," *The VLDP Journal*, 2007.

[6] Xin Jin, "Research on the Model of Enterprise Application Integration with Web Services," in *3rd WSEAS International Conference CEA'09*, 2009.

[7] J. Schmidt, "Enabling Next-Generation Enterprises," *EAI Journal*, vol. 2, no. 7, 2000.

[8] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, 2007.

[9] M. Mecella and B. Pernici, "Designing wrapper components for e-services in integrating heterogeneous systems," *The VLDB Journal*, vol. 10, no. 1, 2001.

[10] Anke Gericke, Mario Klesse, Robert Winter, and Felix Wortmann, "Success Factors of Application Integration: An Exploratory Analysis," *Communications of the Association for Information System*, vol. 27, no. 1, November 2010.

[11] Robert K. Yin, *Case Study Research, Design and Methods*, 3rd ed., 2003.

[12] SEI Team, Open Transaction Framework, 2005.

APPENDIX A – PROTOTYPE CLASS DIAGRAM

The diagram below shows the final class diagram of the prototype developed throughout the case study. Present are the different layers and all classes relevant to the problem. It is the executors that call the integration middleware.