Thesis Project

Architectural Design of Loosely Coupled Services

A Case Study

Author: Magdalena OSBAKK
Supervisor: Jesper ANDERSSON
Examiner: Johan HAGELBÄCK
Semester: VT2015
Subject: Computer Science

Abstract

In the fast moving world of software engineering many are trying to get their piece of the pot of gold. To do that the engineering process needs to be as cost efficient as possible. Since time is money smartly designed systems make full use of already implemented software to save time and money with new development. The usage of services have become a well used strategy for the re-usage of software within as well as between businesses. Loose coupling has long been an architectural strategy for achieving modifiability. The loose coupling between services within a system has lately also been a subject of discussion, since there may be several advantages of using the principles regarding loose coupling and high cohesion within and between services. This document will examine the possible benefits as well as concerns of decoupling already tightly coupled services. As well as discovering patterns and anti-patterns regarding coupling and services using Visma Spcs as a case study.

Preface

This thesis is a case study of Visma Spcs usages of services, in particular of the service inventory Visma Online which is used as a common knowledge base between services as well as applications within Visma Spcs. Visma Spcs have shared their code repositories with me to make this project possible and I am grateful for the trust I have been given.

At times this project may have made me feel like I have bit off a bit more than I can chew, so I would in particular like to give acknowledgement to my supervisor, Jesper Andersson, for all the help and support during this project.

The diagrams in this document is created using the free to use online tool draw.io.

Keywords: Coupling, Services, SOA, API-Gateway

Contents

List of Figures

List of Tables

Abbreviations and terminology

Modifiability - Quality attribute, the level of which a system can be modified without effort.

Interoperability - Quality attribute, the level of which parts of a system can interact seamlessly

RPC - Remote procedure call, a local call that executes on a remote service somewhere.

B2B - Business-to-business, marketing strategy.

WCF - Windows Communication Foundation, a framework for service oriented programs.

HTTP - Hypertext Transfer Protocol is a protocol for communication over the web.

SOAP - XML based communication format/protocol.

SOA - Service Oriented Architecture, a set of principles.

REST - A set of principles, read more at www.whatisrest.com

Service provider - A component offering a service.

Service consumer - A component that consumes a service.

Service composition - A set of connected services

Service inventory - A service inventory is a collection of associated services.

SaaS - Software as a service, a delivery model which includes cloud solutions, not to be mixed up with SOA which is an architecural strategy.

CI - Continous Integration is a practice for integration.

CD - Continous Delivery is a practice for delivery.

LoC - Lines of Code.

1 Introduction

"The Only Thing That Is Constant Is Change -" Heraclitus

This thesis project will describe a case study of the architecture of the service inventory Visma Online. The focus will be towards older parts of the system where old architectural ideas remains and tight coupling occurs at a higher level than the newer projects. Visma Online started being used as a common platform by the desktop applications, such as Visma Administration. Since evolving towards a more service oriented approach Visma Online has become more like a service inventory, where many services used in other projects are gathered.

1.1 Background

In early history of computers programmers placed simple instructions into memory, later this was automated and symbolic names started to be used for operations. This resulted in assemblers followed by macro processors which can be viewed as the first abstractions of software. Higher level languages allowed more sophisticated programs, patterns emerged. Modules and abstract data types were introduced. A shared intuition was emerging among programmers that getting the basic structure right will ease the rest of the development. This intuition became theories. This is all the very beginning of the research of Software Architecture.

But software architecture is not something that was invented long after the first software. The fact is that where there is a software there is also a software architecture, intentional or not [4]. What has been researched and still is being researched is new tactics to achieve a good software architecture.

Visma Spcs is an over 30 year old company that started in 1984 with selling word processing programs [5]. Since then Visma Spcs have focused on Business-to-business(B2B) software and have since 2006 shifted focus towards cloud based solutions.

Both the field of computer science and Spcs have evolved over the years and so has the software architecture of Visma Spcs had to do. As there is virtually no way to predict the future an evolving architecture is perfectly normal. 2013 Visma Spcs had an increased turnover of 90 MSEK in only two years. Visma Spcs has over 150 000 users of their Software as a service(SaaS) solutions, their goal is for this number to increase to 200 000. These numbers just shows that enabling growth and change is something that employees at Spcs has put quite some thought into and is rather important to reach their business goals.

Visma Spcs has evolved towards a service oriented architecture with the goal to achieve a high level of code reusability. Visma Online serves as a common knowledge base used by several applications as well as other service providers in the Visma Spcs system. New applications being built should be loosely coupled from Visma Online and other services, by communication through the APIs. However applications that were built before these steps towards a service oriented architecture were taken remain tightly coupled to Visma Online by direct usage of classes within Visma Online. This has been seen to cause some problems and concerns regarding modifiability, scalability, testability and understandability of the code which is why there has been thought about reorganizing the project Visma Online.

1.2 Motivation

Many companies have problems with old software architectures without any documentation or with poor architectural documentation. Functionality may be duplicated and interfaces tightly coupled. Design decisions and the rational behind them are often lost, which causes risk of erosion in the architecture and the cost of maintenance higher than necessary[6].

Studies have shown that between 50 and 70% of a softwares lifecycle cost is evolving the system, after the first release[7]. Companies reduce these costs by focusing on the modifiability early in development. Architecture is an important foundation for making the system more maintainable and modifiable.

Loosely coupled architecture patterns have been seen to reduce both complexity, dependencies and risks. It can also ease a more agile approach by enabling quick changes and ease maintenance. When making architectural decisions it is important to analyse the advantages and disadvantages of a level of coupling. For businesses in the B2B world loose coupling is often the only approach. Loose coupling does come to the price of more advanced structure which requires a higher competence of the developers but if the system is often undergoing changes the advantages of loose coupling will pay off in the long run [8].

Making these changes in an already implemented architecture can be rather costly but there may be advantages for companies to extract the current architecture, find out where it has erosion compared to the original ideas and principles and redesign parts of the architecture where there are risks.

1.3 Previous research

The field of software architecture is well researched. The fact that a well defined, thought through software architecture reduces both risks and costs of a software project is well known in the field of software engineering. There are methods for doing architecture recovery[9], architecture analysis [7][10], as well as known patterns [11] that are useful in different situations and known anti-patterns [12] which should be avoided in certain situations as well as many strategies with different principles. This report will take up principles and patterns for architectural strategies like SOA and EAI as well as well as well known design principles and patterns for Object Orientation. Previous research will be the foundation for the theories and conclusions.

1.4 Problem definition

For Visma Spcs the most important requirements are:

- Cutting down the code of Visma Online to a clean core
- Support continuous delivery
- Not having to consider the old, smaller services when the platform Visma Online is evolving.

Currently Visma Online does not fully live up to these requirements since some parts are still tightly coupled. Dependencies between Visma Online, Sms, CreditCheck and Backup causes problems with extending, deploying or testing the system which does not comply with the requirements listed above. The problem that needs to investigate is therefore how it can be possible to break the dependencies within Visma Online and the old,

smaller parts of the system to better support Visma Spcs agile process and their constantly evolving system.

1.5 Research question

The objective is to make a tightly coupled service based architecture loosely coupled and describe the advantages of this from a cost perspective.

1. Is there a better architectural strategy for Visma Online services to decrease the degree of coupling?
   (a) Where are the services currently tightly coupled?
   (b) Is it possible to identify anti-patters with regard to coupling?
   (c) What are the problems caused by this tight coupling?
   (d) What design patterns can be used to solve the tight coupling?
2. What are the benefits and trade-offs of a more loosely coupled architecture?

1.6 Expected result

Expectations are that decoupling parts of the current architecture can be of cost benefits for Visma Spcs and any other company with problematic coupling. The expected result from Visma Spcs is a suggestion of improvement of the current architecture of Visma Online, also considering other projects within Visma Spcs and the usage of the services provided by Visma Online.

In order to reach these goals the following steps needs to be taken:

- Look over the current architecture in Visma Online.
- Look into how Visma Spcs in the best way can decouple services that today are tightly coupled with Visma Online.
- Perform a modier analysis. In order to do that the current architecture must be understood and the most important scenarios in order to do an analysis based on where the company is currently going. At the same time the common client library needs to be taken into consideration.

1.7 Scope and limitations

There is no ultimate architecture fit for all purposes. What is a good architecture depends on several scenarios and quality requirements. A good architecture is a trade off depending on the stakeholders needs. So will is it possible to suggest the ultimate architecture for Visma Online? There is no yes or no answer to this question and we will have to accept this. The only question there is an answer to is what is a good architecture for the currently most important scenarios.

The system of Visma, which is much larger than Visma Spcs, is out of scope for this project. Most of the Visma Spcs system is also out of scope and I will provide enough information that is necessary to understand what my work will have to conform to. There is no time or space to describe every single place in the system where some anti-pattern is implemented or principle broken so this report will give an overview of the most relevant current problems and a solution that fits the purpose.

1.8 Outline

The thesis report will be structured in the following sections:

Section 2 - Theory presents the background, previous research and knowledge necessary to understand the case-study. Coupling, Object Oriented Principles, Enterprise Application Integration, Service Oriented Architecture, Microservices, Patterns, Anti-Patterns, Documenting Software Architecture, Software Architecture in Agile Development, Erosion, Recovery and Evolution, Analysis methods, Continuous and Integration will be explained briefly. For a deeper insight in these subjects I would recommend to read the respective sources.

Section 3 - Method will describe the approach taken to the work performed and the method for analysis software architecture done during the thesis.

Section 4 - Case Study of Visma Online will present the work that has been done. The recovery and analysis of the current architecture, some other architectures possible, an suggestion of how to combine those architecture patterns to a new architecture, a comparison of the two architectures as well as an analysis of the work done.

Section 5 - Discussion will in more detail discuss the work performed and the results.

Section 6 - Conclusions will wrap the thesis up by providing answers to the research questions as well as suggestions of what can be done as future work and research.

2 Theory

"A good decision is based on knowledge and not on numbers -" Plato

In this section the background knowledge and theory necessary to follow the reasoning, analysis and the conclusions during the rest of the rest of the report can be found. This section covers architectural principles, patterns, anti-patterns, recovery and analysis methods.

2.1 Introduction

According to Brooks complexity, conformity, changeability and invisibility lies in the very core of software [13]. The complexity is something that makes things like communication, managing, over-viewing and understanding the software system difficult. On top of this the software must conform to the environment, is ever changing and many things like security risks are invisible[13]. These core attributes of software systems still hold and are the very reason for why an elaborate design is important to reduce the risks involved in a software project.

2.2 Theoretical depth

So what is good architecture? A software architecture is important for the success of a software system[4], and yet there is no such thing as a single good software architecture. As Brooks concludes there is no silver bullet for software development [13]. Instead we might describe an SA as being more or less fit for the purpose, or the business goals of the software. But yet there is some hope for evaluating software architectures and creating good SA, with the business goals of the organization in mind [4]. Luckily there are some well defined strategies, principles and patterns that can be used to our advantage as well as anti-patterns known to cause problems in certain situations. Those relevant for understanding the rest of this report will be presented in this section.

2.2.1 Coupling

The ISO vocabulary provides among others loose coupling as "in software design, a measure of the interdependence among modules in a computer program". [14].

The term loose coupling is often used when it comes to both tactics, principles, patterns and anti-patterns within a software architecture. It is important to get a good idea what loose coupling within in the context of software really means. One way of describing loose coupling within software acoording to Bass et al as components that can perform their task independently but communicates when necessary or as programs with interfaces [3].

What might be difficult to define from this explanation is what "when necessary" really means. My definition would be that loosely coupled components knowing very little about the other components, and that components can be replaced without affecting the others.

So why decouple tightly coupled components? The only thing that is certain is that everything in a software system changes. The solutions for this within the subject of software architecture builds on high cohesion and loose coupling. Loose coupling and high cohesion is a tactic to enhance modifiability of a complex system. As Bass et al. so concisely described it: "high coupling is an enemy of modifiability" (121, [4]). This

report will focus mostly on coupling but cohesion must not and can not be forgotten. This case study will examine how much coupling is enough for this given context and what the strategies are for achieving just the right amount of coupling.

2.2.2 Object Oriented principles

There are several design principles, this section will present some object oriented principles that are relevant for the case study in context of coupling and services.

The Single Responsibility principle is by Martin defined as keeping things that change for the same reason together and those things that change for different reasons separated [11]. Simply put: what will change together stays together.

The Dependency Inversion principle states that dependencies should be towards abstractions. This because an abstraction changes more seldom than a concrete class. In Object Orientation interfaces, abstract functions and abstract classes are appropriate to depend upon[11].

The Stable Dependencies principle simply states that dependencies should be in the direction of stability. What is stable does not necessarily need to be decided by what is most frequently changed but can also depend on how much work a change would take [11].

2.2.3 Enterprise Application Integration

Enterprice Application Integration (EAI) is a strategy for connecting loosely coupled applications in an enterprise. Nimmons claims that EAI has come up as a solution to the disadvantages of point to point integrations, see figure 2.3a [2]. The common solution with EAI is using a Hub and Spoke pattern to connect applications, see figure 2.3b [2]. The hub connects applications by standardized interfaces or services. This makes replacing one application easy without having to deal with ripple effects in other applications since the connection is encapsulated in the hub. According to Linssen this solution can be difficult and costly to build but the connections are easy to control since they are centralised [15]. The greatest advantage of EAI is the cost efficiency and that the most important drawback of EAI is the fact that it can cause problems in scalability in the long run [15].

2.2.4 Service Oriented Architecture

Service Oriented Architecture, SOA, is a well used strategy to achieve loose coupling and enable reusability. Since Visma Spcs are using services and the main goal of this study is to define if and how the services can become more loosely coupled, this is an important concept. The terminology that will be used in this report when discussing about services and service orientation will be presented in this subsection.

Service providers, service consumers, service compositions and service inventories will be mentioned throughout the rest of the report. A service provider is a component offering a service while a service consumer is a component that consumes a service. One component can be both a service consumer and a service provider. A service composition

is a set of connected services. A service inventory is a collection of associated services [1].

SOA is not attained merely by using services, nor can it be achieved only by using specific technologies. SOA is an architectural strategy and there is more to be said about SOA than what will be taken up in this report. The core principles are: Standardized Service Contracts, Service Loose Coupling, Service Abstraction, Service Reusability, Service Autonomy, Service Statelessness, Service Discoverability and Service Composability[1].

These principles aim to achieve loose coupling which in turn enables modifiability, interoperability and integration. SOA is applicable at any level of a business, within a department or an entire enterprise. However SOA gives the greatest value at enterprise level [16]. As in any architectural strategy there are disadvantages as well as advantages of using SOA. The trade-offs includes the complicated design and implementation as well as loss of performance due to the communication channel. Networks may not be stable so if not carefully designed reliability can be affected, as in any distributed system.

2.2.5 Microservices

Newman introduces a term for SOA where everything from Software Architecture, System Architecture to Enterprise Architecture has a high level of loose coupling and high cohesion [3]. This builds on the idea of small teams maintaining small services. The services are autonomous, loosely coupled with high cohesion and reflects the business goals of the organization. Newman concludes that loose coupling and high cohesion should comply as well within and between and within services as in any other SA tactic. This all in line with known software design tactics and principles as well as SOA principles [3].

Microservices in itself is not a new idea, what is new is the term microservice. The principles of low coupling and high cohesion is not big news either. The ideas behind this term has been derived from the real world and how SOA is often used when done well. Some of the ideas that are the foundation of the term microservices are domain-driven design, continuous delivery, on-demand virtualization, infrastructure automation, small autonomous teams and systems at scale[3]. What we can take with us from microservices is the idea of services being loosely coupled from other services instead of viewing a service inventory as one large monolith.

2.2.6 Patterns

SA patterns is not something that is invented but rather discovered[4]. The definition of patterns is something found to be used in many cases to solve a specific problem, in this case the loose coupling between services. The usage of already discovered patterns may help us in our work with finding a SA for our purposes. There are many patterns for different views of the architecture and to achieve different quality attributes[4]. What is important to remember is that in many cases patterns are not something where we must choose one or the other but instead combine them in order to get the desired quality attributes for the software in question.

Layered pattern is a commonly used pattern for attaining smaller less coupled modules. The number of layers as well as the number of layers using the same layer is not specified for this pattern. The tell-tale of a layered pattern is that the modules are separated into groups, what we call layers. Relationships between layers are only allowed to go one way, also called unidirectional relationships. The layers are cohesive and encapsulated[4]. See figure 2.1 for a representation.

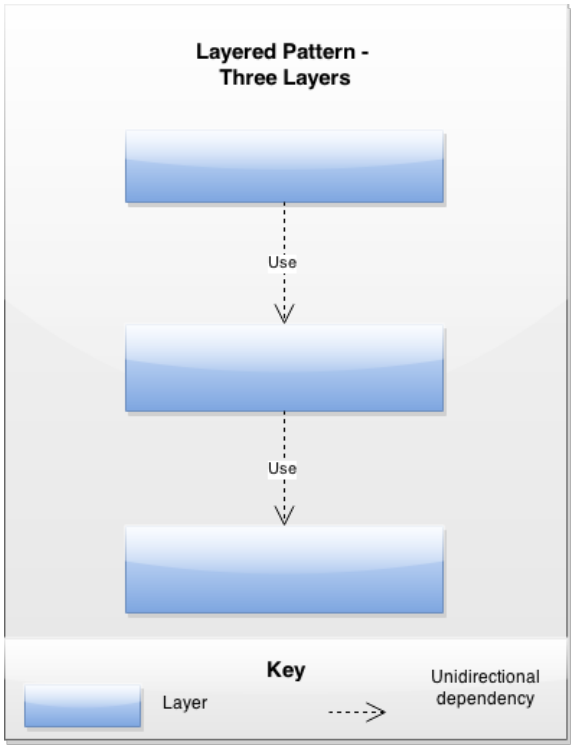


Figure 2.1: A diagram of the Layered Pattern, in this case there is three layers.

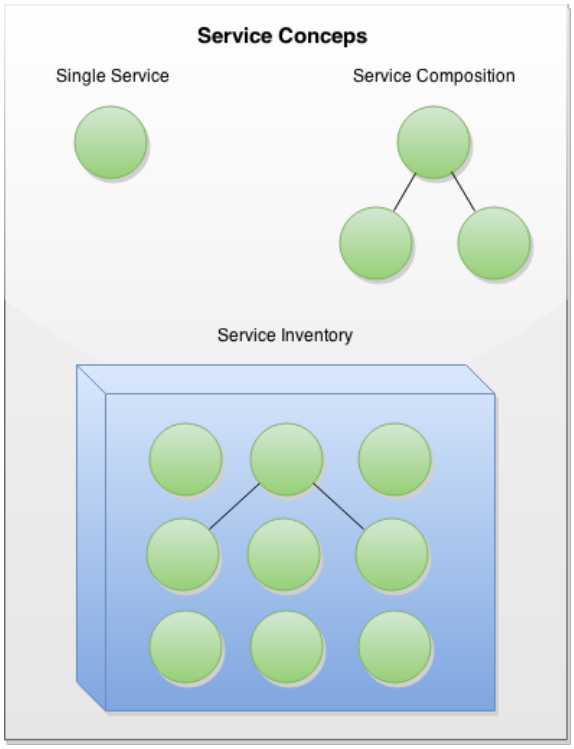


Figure 2.2: An Abstraction of common service terms, Source: [1]

The advantage of using layers is that the layers can be implemented separately[4]. The trade-offs include the extra cost of design and loss of performance in the case that information needs to go through one or more layers before being used[4].

The Multi-Tiered pattern can be used both as a component and connector pattern and an allocation pattern[4]. In this context it will be used to describe the distribution and allocation of the components to specialize the deployment towards loose coupling. A distributed system may need to be split up into component groups with independent executions. These structures are all represented by a tier. Connectors are only allowed within components of the same or adjacent tiers[4]. The main difference between the tiered and the layered patterns is that a layered pattern is a static view where a layer is a group of modules, while the tiered pattern is a dynamic view where a tier is a group of components. The disadvantages of this pattern is mainly the initial cost, for a smaller system the cost might not outweigh the benefits[4].

The Hub and Spoke pattern is one solution to complicated point-to-point communications, see figure 2.3b. The hub and spoke pattern gathers the communication to one component, making the other components not aware of each other[2]. This leads to less coupling when it comes to knowledge of other components, but there is a risk that as the system grows the hub carries to much responsibility, possibly this could be a bottleneck of the system[2].

Enterprise Service Bus is a design pattern that handles complex relations between components[17]. The Enterprise Service Bus (ESB) is a virtual bus with the goal to standardize the communication between applications by connecting applications through standardized interfaces or services[15]. The ESB solves problems with standardizations and point to point integration, however the applications needs to speak the same language as the bus causes difficulties in connecting new applications[15]. When using a strict version of ESB the applications connecting to the bus must use the same protocol as the bus and the ESB routes the messages. The disadvantages of ESB is that this approach has at times been proven to be time consuming, costly and complex[15].

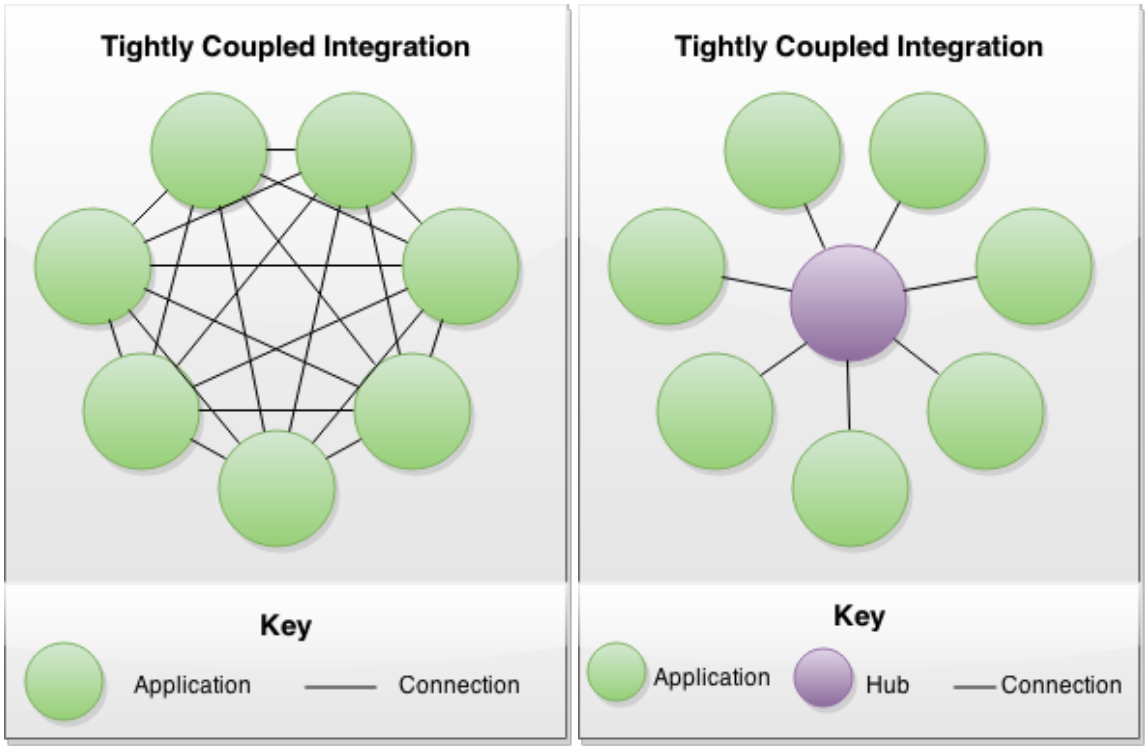
The Broker pattern is used in a distributed system so that service consumers, or clients, do not need to know details like locations of the service providers or servers. A broker in this context is a run-time component that handles the communications between service providers and service consumers[4].

The main advantage of using this pattern is that by avoiding point-to-point communications the consumers are unaware of the characteristics of the providers. The broker can be designed in such a way that if one server, or one service provider, is not available it can be replaced by a compatible component and only the broker contains the logic surrounding this redirection. This is a pattern that enables interoperability and orchestration[4].

The disadvantages includes the added complexity, latency in communication between service providers and service consumers, difficulties in debugging and if not designed carefully a security risk and a performance bottleneck [4].

2.2.7 Anti-Patterns

An anti-pattern describes a commonly used solution to a problem that is known to have negative effects[12]. In this section the anti-patterns that are relevant to the case-study



(a) Tightly coupled integration/point-to-point (b) The Hub and Spoke Pattern

Figure 2.3: Source: Steve Nimmons [2]

will be presented. The focus will be on patterns regarding coupling, integration and the usage of services.

Point to Point / Tightly Coupled Integration is a pattern that legacy application integration or EAI has been known to use. This approach to integration causes a mesh ending in a brittle, complex structure. When it come to the level awareness of other applications the system is tightly coupled. The results of this anti-pattern are among other a high cost of change, invasive integration, limited abstraction, difficult systems to manage, monitor, scale, secure and troubleshoot. The solutions are among others the Hub and spoke pattern (see figure 2.3b), Enterprise Service Bus pattern, and the Canonical Model pattern.[2]

Using SOA Silos have advantages such as enabling a quick way of getting into SOA. But they also cause problems like redundant work, risky life cycle management, inconsistency in areas like business rules and service contracts and an overhead from areas like duplicate testing and documentation[16].

Without an enterprise view for services you will not get SOAs full potential. If application silos are merely replaced by service silos where each development team develops their own set of services there is risk for a "service sprawl", which prevents the economical advantages of reusing services [16]. Silos are contradictory with several SOA architectural principles such as the principle of having explicit contracts between service producers and service consumers.

The solution can not be based on merely technology. The solution is according to Heller communication processes or in other words to share and design services, architectural principles and artifacts over the enterprise[16]. Heller suggests a center of excellence or architectural board to avoid this type of pattern to occur and ease the transition

to SOA[16].

SOA equals EAI 2.0 is an anti-pattern since many companies views SOA as "Web services managed with an enterprise service bus"[16]. Which in that case would mean that SOA is merely a new way of doing the things already possible with EAI, hence the name EAI 2.0. The problem with this is that SOA offers more and treating SOA as EAI 2.0 hinders full use of SOAs advantages that one could achieve by following the principles of SOA[16].

Vendor Lock-In, Reinvent the Wheel, No Legacy and All from Scratch Vendor Lockin is an anti-pattern within SOA that is also known from Object Orientation as reinventing the wheel. The failure of reusing legacy systems and third-party products has many names. This failure can limit benefits of SOA as well as other architectural styles. One of the key benefits from applying SOA is actually the possibility of easy integration. In the world of SOA these type of anti-patterns are therefore rather costly. Constantly rebuilding already existing systems is a disadvantage from a cost perspective because of the unnecessary development cost[18]. These anti-patterns do not help when it comes to the level of coupling but rather gives an idea of the fact that one should not rewrite an unnecessary amount of code if there already is functional logic that can be encapsulated.

According to Král and Zemlicka there are advantages to leaving some legacy systems since they can be very stable and have useful capabilities[18]. By choosing an appropriate way of integrating the legacy systems with the new systems the old systems can be reused, saving investment of implementation costs. The integration should comply with the software engineering principle of information hiding or encapsulation and wrap the legacy systems in a way that their interfaces do not need be massively changed. Legacy systems that do their job and are not broken can in this way avoid being changed. Service orientation is suitable to re-factor Stovepipe Systems and is the best known approach to decentralize enterprises[18].

Sand Pile or Fine-Grained Services is a common implementation of SOA. which may be described as "one elementary service per one software component" (10, [18]). The problem with this technique is that the result is many small components that share data sources which cause inefficiency and maintenance problems[18].

The solution to this anti-pattern is to group the related services into one composite service[18]. This has been taken up previously as the service composition pattern[18].

2.2.8 Documenting software architecture

After describing these strategies and patterns that may be used in order to achieve our goals one question that has become important for the work surrounding this report needs to be addressed: why should one document software architecture? It may seem like a waste of time for competent personnel to spend time on documentation, when the architecture should be implemented and then the implementation can serve as documentation. As described by L. Bass et al. even the best of architectures serve no purpose if the stakeholders are not aware of it and if it is not implemented correctly [4]. Not to mention what will happen with an architecture that is all in one architect's head if that person leaves the project. The architecture should be fit for education, communication between stakeholders and blueprints for building the system[4]. Understanding the entire reasoning and

ideas behind an architecture by looking at code is just not possible in a large complex project.

2.2.9 Software architecture and agile development

When using an agile process documentation may be put aside and seen as less important than the implementation. The incremental approach would mean that design documents could change from iteration to iteration. Within an agile project the question is not about doing or not doing an architecture but about how much of the architecture should be documented [4]. The answer to this question is that you should document just what will be needed for the future. Document what new employees will need to know, document the crucial parts of the architecture and the parts that are often changed [4].

2.2.10 Software architecture erosion

One of the risks of not providing sufficient documentation for the architecture and design of a system is that the actual architecture becomes architecture becomes more and more distant from the original ideas. Object Oriented (OO) applications are becoming more and more complex at the same time as the pace of development is expected to keep increasing. This may cause several problems during the lifecycle of an OO application and these problems can be traced down to architectural erosion.

"The architecture giveth and the implementation taketh away"(p.26, [4]). This citation might serve as a fair description of what can go on in a software project. Implementation decisions can break a good design, decisions at all levels will affect the final architecture quality not just the conceptual ideas of the architecture[4].

Martin describes the erosion from a clear and beautiful design to a decaying mess because of ugly hacks being made until making the smallest change is unmanageable [11]. Rapid decisions, stress or always looking for the easiest answer may be some reasons for software quality decaying[12].

2.2.11 Software architecture recovery and evolution

If architectural erosion has gone too far and the original architecture no longer holds for the system, unintended or even unknowingly, performing a software architecture recovery may very well be the only way to really know what the current architecture actually looks like. Not knowing what the current state of the software architecture is makes evolution of the architecture difficult, at least without possibly causing more damage.

The process of backward engineering the current software architecture can serve several purposes. The architects can see if the implementation has followed the principles and patterns set out by the architecture and in that case see if there is any reason to redo some of the implementation to avoid erosion. Another no so unlikely case is that it was never any design in the first place and due to quality issues the software needs remodelling.

There are several methods to do architecture recovery. One method to do software reconstruction as well as evolution simultaneously and iteratively is Focus. Ding and Medovic claims that their method shows promising result to make fast and dependable evolution of software architecture [9]. Focus is appropriate when there is little or no documents regarding the current software architecture. The main ideas behind the method is that software recovery is only one step towards a better software architecture as well as only making changes in part of the system that is going to be worked on. This makes it possible to recover the software architecture at the same time as evolving the software

architecture. This makes the recovery incremental, instead of recovering the entire architecture at once.

2.2.12 Software architecture analysis

Analysing the existing architecture can be one way to discover if desired quality attributes are not currently fulfilled or to discover quality attributes for the a future version. Another reason can be that there are several candidate architectures that need to be compared before choosing between them.

Software and analysis is a well researched subject where there are several methods to choose from depending on how they fit in with the companies and the software business goals[19][7]. One well established and tested method is ATAM which can be applied to all quality attributes[4], others like PASA[19] or ALMA[7] target specific quality attributes.

ATAM - In the Architecture Tradeoff Analysis Method you use scenarios to analyse the architecture. This analysis method is designed so that the person evaluating the architecture does not need to previously be familiar with the software architecture. This method involves several stakeholders including architect, product owners, people making decisions as well as an evaluation team. Usually it takes a full day or more from all the stakeholders involved in the evaluation and more time for the evaluators, which makes it a rather costly method [4]. There are different types of scenarios explored in ATAM. Indirect growth scenarios represents future changes that are expected. Indirect exploratory scenarios represents future changes that is not expected to happen. These types of scenarios are rather similar to the scenarios evaluated in ALMA [7].

ALMA - Architecture-level modifiability analysis is used to analyse modifiability by finding and evaluating change scenarios, in that way you can get an image of how well the system is prepared for change. There are five important steps in ALMA [7]:

1. Choose a goal with the analysis. There are three goals to choose from the ALMA method: risk assessment, maintenance cost or comparison between architectures. Depending on which goal is chosen the following steps will be carried out differently.

2. Architecture description. The goal of this step is to decompose the system into components and the relationship between those components as well as the relationship to the environment of the system.

3. Find relevant change scenarios. The process involves conducting interviews with relevant team members finding and selecting the relevant scenarios.

4. Evaluate the change scenarios from the third step. In this step relevant team members collaborate to determine what impact the scenarios can have on the system. For each scenario the affected components are identified, the effect on the components and the ripple effects are determined.

5. Interpret the results and draw conclusions from them.

Relating the steps for the goal of predicting maintenance cost means focusing on the cost of maintaining the system when going through these steps[7]. The change scenarios need to represent events that will occur in the future. The exploratory scenarios in ATAM are similar to the scenarios looked for when the goal is to predict maintenance cost.

2.2.13 Continuous Delivery and Continuous Integration

These are two concepts that are important for Visma Spcs that must be supported in the new architecture of Visma Online. To get an understanding of these requirements there will be a brief introduction to what we really mean when speaking of Continuous Integration (CI) and Continuous Delivery (CD).

CI has the main goal of keeping every newly checked in code in sync with the rest of the code. This is usually done with some CI tool that once code is checked in, checks it out and carries out testing to make sure the integration was successful[3].

With CD we mean that every check in should be treated as a possible release. This can be done using different stages to each build, also known as pipelines, where one stage is smaller tests for which you can get fast response to the quality of the committed code and the slower tests which are necessary for a release is another stage in the pipeline[3].

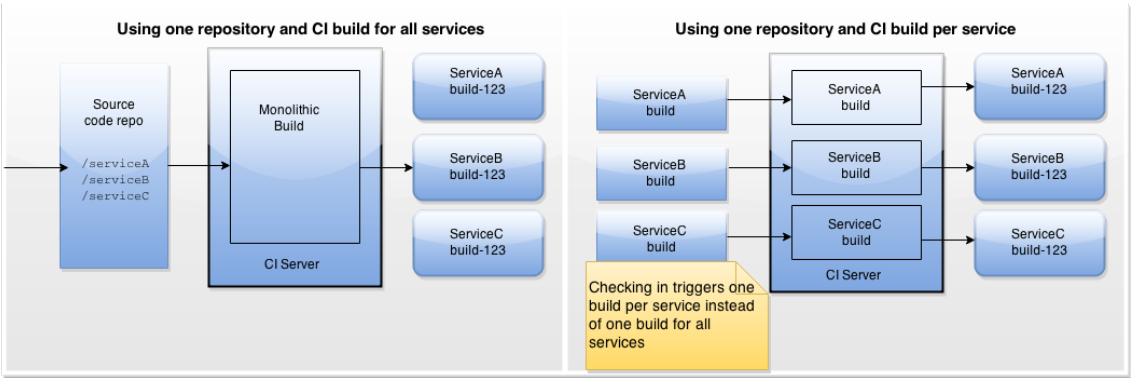


Figure 2.4: Image Source [3], comparison of having different code repositories and one build per service or one repository and one build for all services when it comes to Continous Intetegration.

3 Method

"Every discourse, even a poetic or oracular sentence, carries with it a system of rules for producing analogous things and thus an outline of methodology" - Jacques Derrida

This section will describe the methods used during this work, from outlining the meaning of a case-study to more specific methods that has been used to recover the current architecture, analysing the software's architecture and finding new suggestions for the architecture.

3.1 Methodology

This subject was suggested among others by Visma Spcs. I found this interesting because it included getting a deep view of their code base and finding patterns within existing code. The work has mainly been to perform a case study which is a qualitative method. [20] The case study will be a part of the service inventory Visma Online. An analysis of the current architecture and the most important scenarios given by the supervisor at Visma Spcs will also be carried out.

A literature study will also be a big part of the methodology both to decide what type of architecture analysis will need to be carried out, deciding out what the risks of the current architecture is and what the benefits and trade-offs of a new architecture would be.

Interviewing employees at Visma Spcs, working in the Visma Online project, is a great part of the case study as well. It is necessary to recover the current architecture, analyse the architecture and identify relevant scenarios for the new architecture.

3.2 Scientific Approach

A case study has five major steps: Designing the case study, Preparation, Data/Evidence collection, Analysis, Reporting[20]. A case study is a flexible methodology and the steps are iterative and can be done incrementally. There are some limits to the flexibility since a case study should have the objectives set from the beginning[20].

There can be several different goals in a case study: exploratory, descriptive, explanatory, or improving[20]. This case study will explore the Visma Spcs system architecture, explain how it looks and why as well as give a suggestion for improvement. Surveys, interviews and observations in order to collect data will be used in order to carry out this case study.

3.2.1 Selection

The selection of data will ultimately come down to what is most relevant to the research questions. Visma Spcs has a large code base and the part that examined in this project is the code of the Visma Online project. Within this part of the system there are multiple interesting Software design principles and design patterns used and broken. Some principles are broken for good reasons which is also interesting research. This is all part of the real world. Even if my research has gotten me on many different paths and I have seen and read interesting things from multiple perspectives, this thesis has an outline and that is outline which will be the key to what information chosen to present.

3.3 Analysis

As described there are many methods already published to perform an analysis of a software architecture [4] [10] [19] [21] [7]. For the software architecture analysis ATAM was a good candidate due to it being well established [4]. For the scope of this project ATAM appeared to be a too heavy analysis method since the purpose of the case study was not only to get a deep enough analysis of the architecture but also suggest a new architecture.

Therefore a small survey with the key stakeholders in the project to try to sort out the most important non-functional requirements was handed out. The employees got to answer a survey where the questions were different non-functional requirements and the options were a number between 1 to 5 describing how important this requirement was from their point of view. The result of this survey suggested the focus should be on modifiability which lead to the conclusion of using ALMA as a subset of ATAM but with focus on modifiability.

3.4 Reliability

Trying to estimate how much effort a change in the software might take is in no way reliable. The only way to know for sure what effort it will take is to make the change. When estimating the ripple effects of a change even the most experienced developer or architect can be significantly wrong [7].

3.5 Ethical considerations

There are some ethical factors to consider when doing a case-study. These key ethical factors of a case study includes: informed consent, approval, confidentiality, handling sensitive information, feedback [20].

My supervisor at Visma Spcs as well as other employees approved and gave feedback to my project plan before starting on the project. Before getting access to the code and any sensitive information I signed a disclosure agreement. We agreed that any information published in the final report should be approved by employees at Visma Spcs with competence in the specific areas that could be affected.

4 Case Study of Visma Online

"If you look for perfection, you'll never be content." Leo Tolstoy, Anna Karenina

In this section the current architecture will be explored and analysed, a new approach will be suggested based on the most important scenarios. After that the two will be compared based on those scenarios. There is no such thing as a perfect one size fits all software architectural strategy. There are parts of the current architecture that gives some disadvantages but as that is inevitable in such a complex system, there will be disadvantages as well as advantages to another approach as well. At the end of this section the hope is that these advantages and disadvantages can be weighted against each other to make it clear which best fulfills the most important scenarios in this case.

Visma Online started out as a common knowledge base for several applications in the Visma Spcs system. The applications used information from Visma Online with direct references causing them to be built on top of Visma Online, which in turn caused them to be built and deployed together. When realising that the amount of code that needed to be grouped together would sooner or later grow to a massive unmanageable scale the idea of applications integrating with Visma Online through APIs instead came in mind. With the goal to keep moving towards a service oriented architecture without application silos newer services as well as applications have restrictions to communicate with APIs making the services loosely coupled from each other.

Applications that were already developed when the decision of changing architecture was taken however remain as tightly coupled application silos within Visma Online. These applications are stable and no longer changed frequently, causing them to not be updated to follow the new principles. Visma Online on the other hand is still a growing project. As Visma Online continues to grow having to consider the older applications seem more and more tedious and time-consuming. On top of this the current architecture uses a wild mix of service principles and techniques, mainly because changing the old projects is not prioritised when switching to a new architecture or technology. This also adds complexity to the service consumers.

4.1 Approaches for reporting results

The results of the current architecture will be described in the order of carrying out a software architecture analysis using the method ALMA. First the goal will be described then the results of the software architecture recovery then the results of the change scenario elicitation and the results of the change scenario evaluation. The new architecture suggestion will be presented in the order of the steps that lead to the conclusion of the suggestion. After presenting the suggestion a comparison of the current architecture and the suggested architecture will be presented. Finally an analysis of the case-study and the two architectures will be presented.

4.1.1 Non-functional requirements survey

The result of the survey was that several team members in the Online project thought requirements such as modifiability, extensibility and maintenance that had to do with the future changes of the system being cost efficient was between 3 to 5 on the scale of 5. This was the expected result and gave support to focusing on modifiability in the upcoming analysis.

4.1.2 Software architecture analysis method

The survey result described in the previous section motivated the choice of method to use for the analysis of the architecture to be ALMA, which is an analysis method that focuses on modifiability.

The goal of the analysis has been, as requested from Visma Spcs, to be the maintenance cost. Risk assessment would also be an interesting viewpoint. The relevant parts from ALMA will be used to assess maintenance cost and then do a comparison with the new architectural suggestion. Having risk analysis as a goal was also an option, however the costs was of more interest to Visma Spcs. The risks will still be reasoned about but the maintanence cost will be the main focus during the comparison of the two architectures.

4.2 Analysis of current software architecture

Before analysing the current architecture or giving a suggestion of new architectures there needs to be some understanding of the current architecture. Software architecture can many time shift from the original idea, or evolve into something else so software architecture recovery might need to be done even if there is documentation. Sometimes there were none, or not sufficient, documentation to begin with. This is where this work needed to start. The SA was described in relevant views to the project and here follows a subsection for each relevant view with description and the advantages and disadvantages of the approach taken.

4.2.1 Service View

This view is intended to give insight to the way Visma Spcs has adopted the service oriented approach to software architecture. Visma Spcs usage of services can be seen differently depending on which way we view it.

Visma Online is intended to be a service inventory, containing much of the gathered logic of the system. The idea is that this is the place where logic that is common between different applications recite. Visma Online provides services for other applications, see figure 4.5.

Even though the approach for providing services has changed over the years, using RPC with SOAP or WCF, using Service Bus or not and most recently adopting RESTful web services, see figure 4.6, the central role of Visma Online remains. However an approach that looks less like the hub and spoke pattern has been evolving, where each service that has a clear business goal is loosely coupled from Visma Online. One example of this is the Mobile Scanner service which serves as a service provider towards Administration at the same time as a service consumer of services from Visma Online.

It is a clear trend in Visma Online's architecture that the older techniques decided not to be built anymore are not replaced but rather remains as legacy code. Legacy applications such as the desktop application Visma Administration have contracts to Visma Online's SOAP services. Some of the service oriented components such as EAccounting is using WCF while the newest solutions most often use RESTful services. One example is the MobileScanner mobile application which uses a REST API to connect to Visma Online and also has its own REST services between the frontend and backend and for other applications to use.

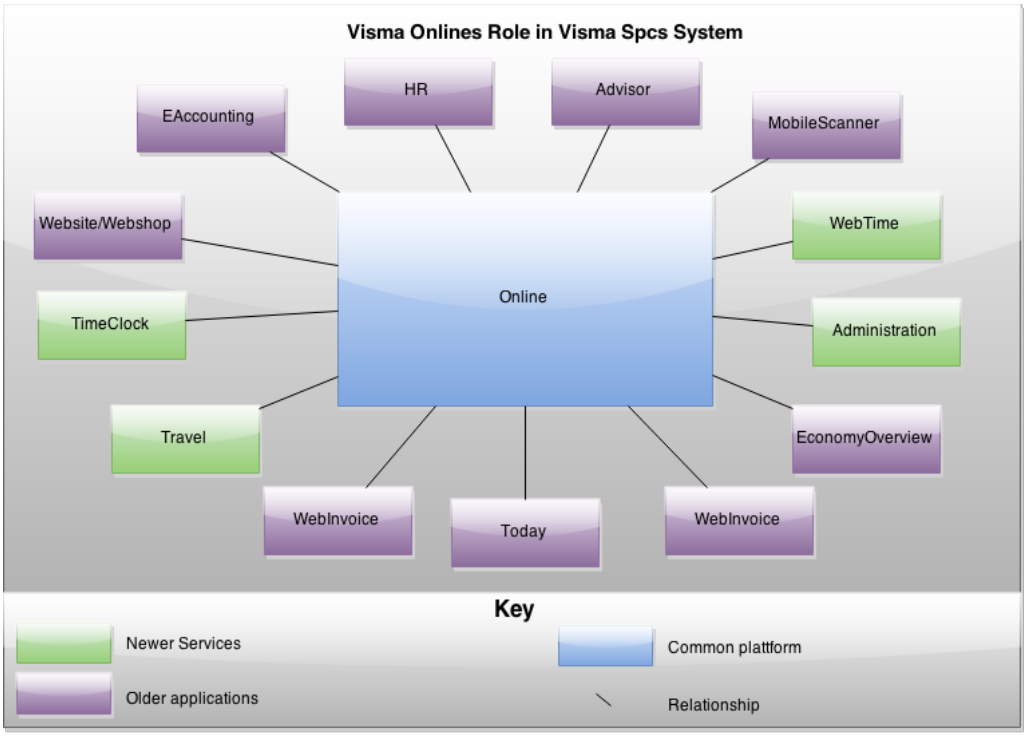


Figure 4.5: Diagram of Visma Online as a common knowledge base/service inventory providing services to other applications in Visma Spcs system. From this point of view there are similarities to the hub and spoke pattern.

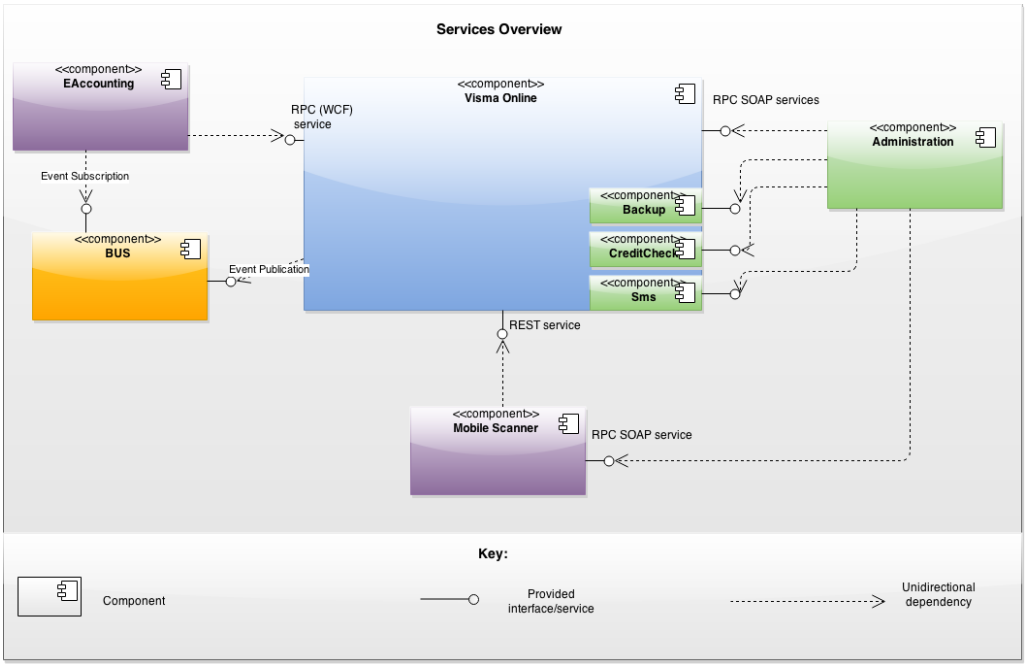


Figure 4.6: Component Diagram of the different approaches used for the services of Visma Online. There is a clear trend that code already built is not changed to follow the new principles but rather remain as they are, while the new code follow the new principles.

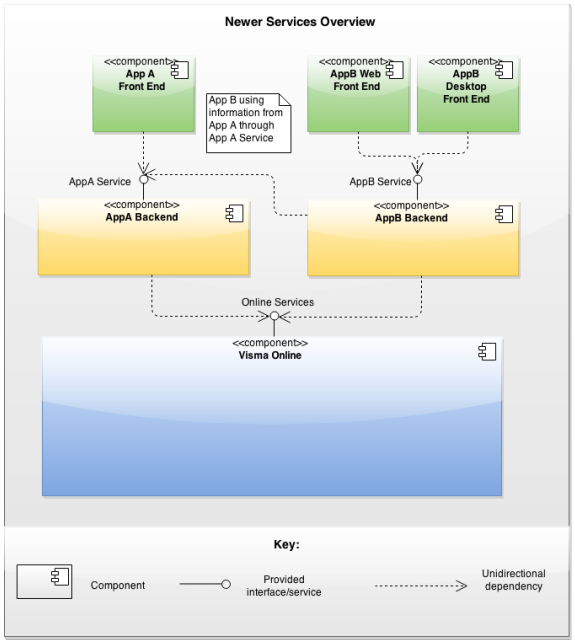


Figure 4.7: Abstraction of the ideas of the newer services. Each component achieves loose coupling from other components by using services, the services are usually reflecting a reusable business goal. If for example Application B requires information from Application A the information must be passed through the service. As previous diagrams also shows the connection to Visma Online goes through Visma Online's services.

Newer Services Simplified View In some cases services are used between the backend and the frontends of the applications as well as between Visma Online and the applications, see figure 4.7. From this point of view the applications are not application silos merely connected through the platform, but instead each applications backend is in fact a service provider. This because both the web frontend and desktop application frontend connects to the same backend to avoid redundancy. In theory the frontend of the applications and the backends could with this approach be deployed separately and the risk and cost of changing the view or the backend is very small as long as the service contract remains the same, due to the fact that the changes are isolated to relatively small modules.

From this view the step to the idea of Microservices is not far. Instead of viewing the architecture as applications and a common plattform we may view it in a service oriented way, where each application backend is a service built on a specific business context. EAccounting is one business context and therefore one service. Visma Online is however still a significantly larger service than what is justifiable within the context of microservices.

Services Deployment View The idea of the current SA is that each service should be able to be deployed separately since they are communicating through services. The backends of the applications and the frontends should if built separately be able to be deployed separately, see figure 4.8. This is similar to the tiered pattern where each loosely coupled component should be able to be deployed on a different host if it becomes necessary. The communication channel between the tiers are the APIs. One example of the usefulness of this pattern from Visma Spcs point of view is that applications can be moved to a cloud host, in the case of Visma Spcs Azure. EAccounting is one application that while my study began was currently in the process of being moved to Azure.

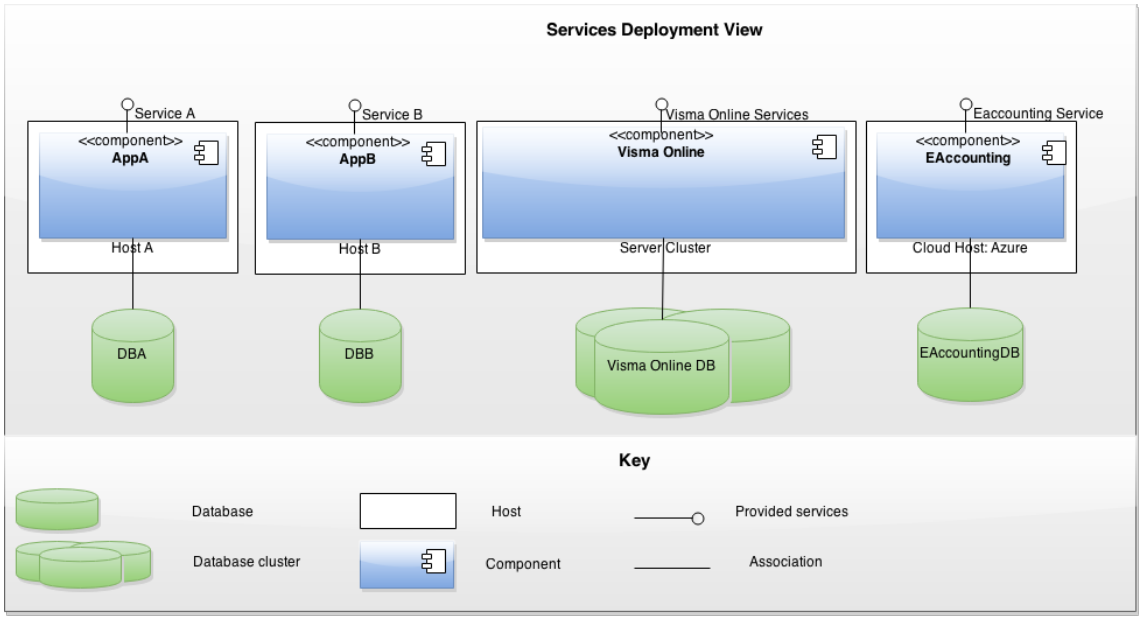


Figure 4.8: Component Diagram of the services deployed. For applications only communicating through services it is possible to be deployed on different hosts. EAccounting is one application that is currently being moved to Azure.

Advantages of this approach is that the cost of hardware can be reduced, instead the cost can more easily be customised depending on how many users the service has. Visma expects EAccounting to have a significant increased number of users by the end of this year so this flexibility has been proven to be cost efficient. The negative effects represents those of a the tiered pattern such as networks not being reliable, the extra cost of the design and the performance drawback due to the communication channels.

4.2.2 Application Relationships View

This view is intended to show the relationships between Visma Online, Sms, Backup and CreditCheck in more detail. This view shows in more dept where the scope of the project lies. In the view described in the previous section the tight coupling appears to not be a problem. However the ideas of the previous section is not implemented in the entire system, i.e. not all applications communicate with Visma Online through services. Visma Sms, Visma Backup and Visma Credit Check use direct references, causing dependencies, to communicate to Visma Online's libraries. Due to this tight coupling it is not clear where Visma Online ends and Sms, Backup and CreditCheck begins since they do not have a clear encapsulation and interfaces between Visma Online. Rather these components will be represented as being inside of the Visma Online Component, see figure 4.9. However these components still do have services and other applications are not allowed to communicate to them through other means than these services, see figure 4.7. Visma Administration is communicating to Visma Sms, CreditCheck, Backup, Online as well as other services through SOAP based contracts.

In the sense of communications to Visma Sms, Visma Backup and Visma CreditCheck these applications do follow the principles of the rest of the Visma Spcs system. However in the communication between Visma Online and Visma Sms, Visma CreditCheck and Visma Backup the standard is not followed but instead they are tightly coupled with Visma Online. From a static view, see figure 4.10, this can be better depicted. Since Visma Sms,

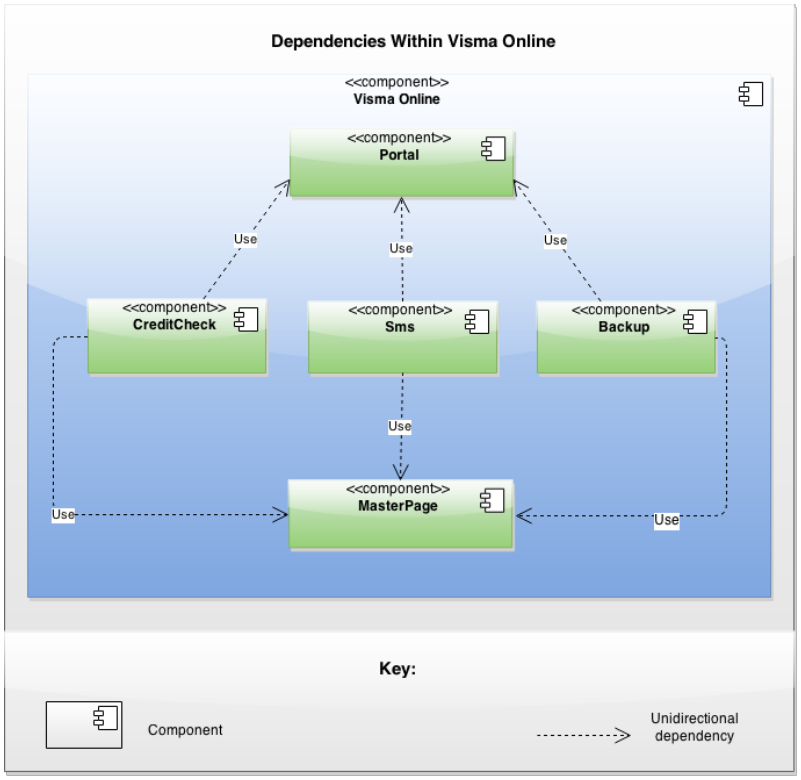


Figure 4.9: Component Diagram representing the coupling between components of Visma Online.

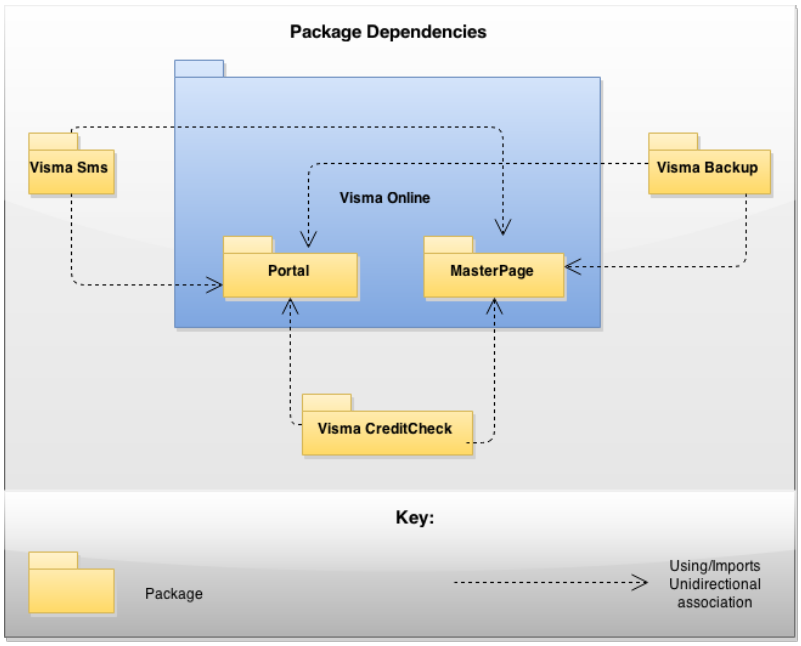


Figure 4.10: Package Diagram representing a static view of the dependencies between Sms, CreditCheck, Backup and packages of Online.

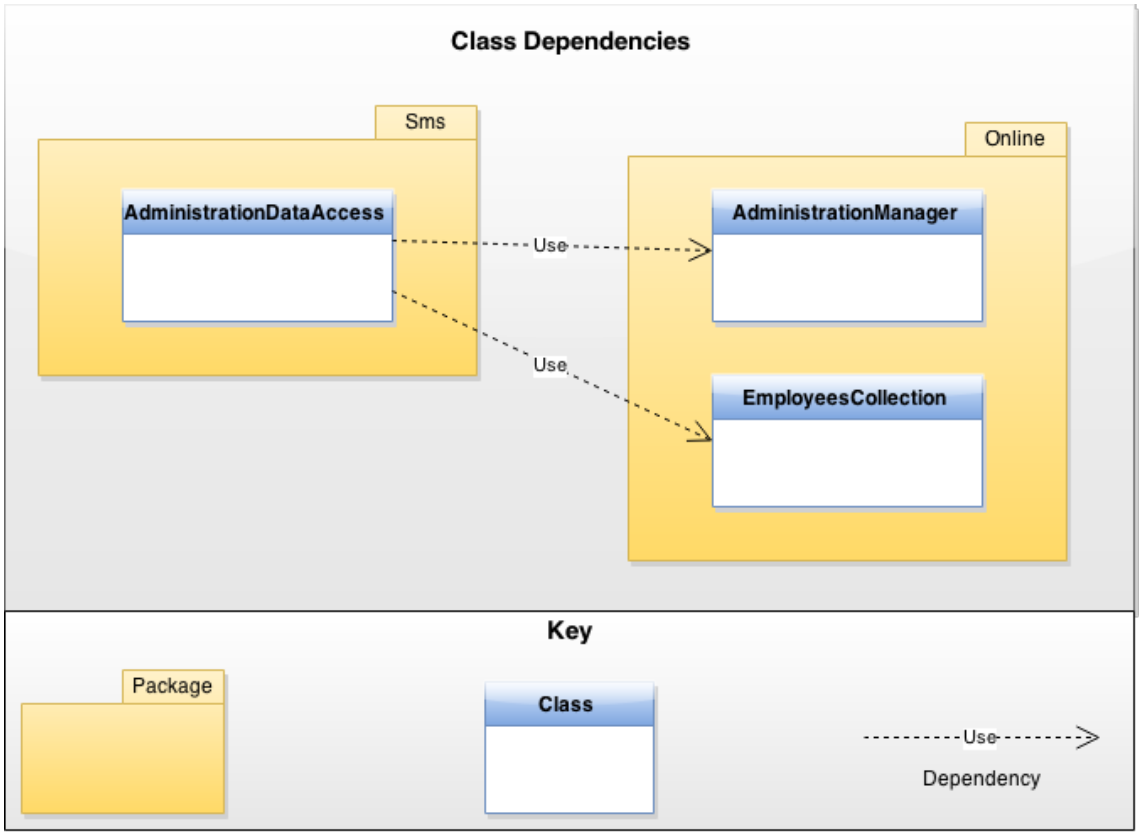


Figure 4.11: Visma Sms and Online coupling, Class Diagram of one example where there are problematic coupling between classes in Sms and classes Online. This class diagram is highly simplified and only contains what is necessary for my line of reasoning. This is not the only place where the dependecies looks similar but serves as an example of the problematic patterns that can be found.

Visma CreditCheck and Visma Backup depend on Visma Onlines packages Portal and MasterPage, which is the main cause of this tight coupling.

The dependencies between Visma Onlines packages Portal and MasterPage and Visma Sms, CreditCheck and Backup cause the deployment of all these four applications to be at the same host, see figure 4.13, in this case a server cluster. The difference between other applications that only communicate through services is that the dependencies needs to be broken before Sms, CreditCheck or Backup can be moved to another host, for example Azure where some of Visma Spcs applications currently are hosted.

Disadvantages of this approach Visma Sms, Backup and CreditCheck are application silos that recites within the plattform causing problems of application silos that SOA is intended to solve. SOA therefore does not live up to its full potential. With this approach Visma Online becomes a larger code base, which eventually may be to large to handle. This tight coupling also leads to all these applications needing to be built and deployed together with Visma Online. As well as not only the space of one application can be scaled, which may lead to unnecessary hardware cost.

The only advantage with keeping this architecture is that Visma Sms, Visma Backup and Visma CreditCheck are stable applications which do not need much maintenance. So not changing them now will save time at the moment.

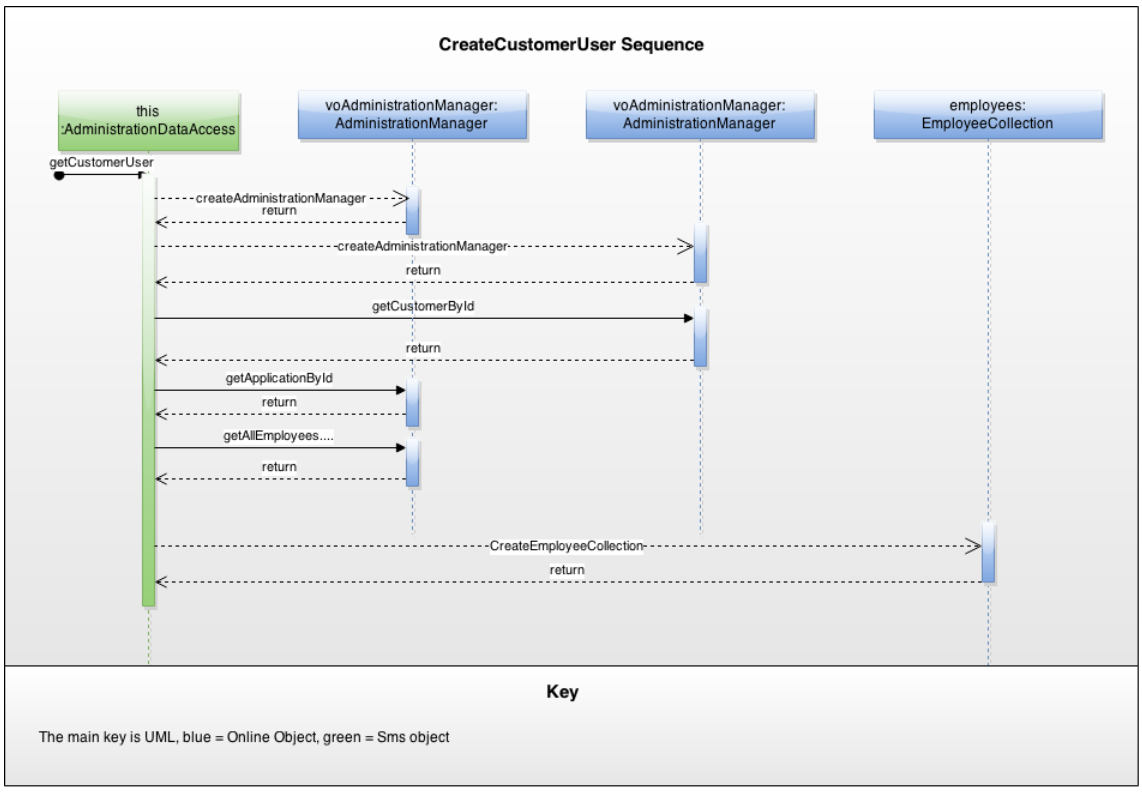


Figure 4.12: Visma Sms and Online coupling, sequence diagram of one example of where there is coupling between Sms and Online. This Sequence diagram is highly simplified and only contains what is necessary for my line of reasoning. This is not the only place where the coupling looks similar but serves as an example of the problematic patterns that can be found.

4.2.3 Internal Package Views

This view is mostly intended to serve as reference for the analysis. The packages will be discussed when describing what code may need to be changed in the change scenarios.

Sms, Backup and Creditcheck are all based on the layered pattern, see figure 4.14. Both The View and the Services have a tight coupling towards the logic layer, but the logic layer has no knowledge of the layers above.

The static diagrams do not need to be detailed but merely provide an overview of the packages in the application to provide better understanding in the following sections. Classdiagrams where also created, which due to the large amount of code became pages long with more information then necessary to carry out the project. Therefore they are excluded and these simplified package diagrams are enough, see figures 4.15, 4.16 and 4.17.

4.2.4 Discovered Anti-patterns

Anti-patterns are patterns known to cause certain problems, and this is one thing we can look for to see if we can identify some future problems. The anti-patterns found in Visma Online, Sms, Backup and CreditCheck were:

- SOA with application silos [16]
- Using SOA and EAI 2.0 [16]

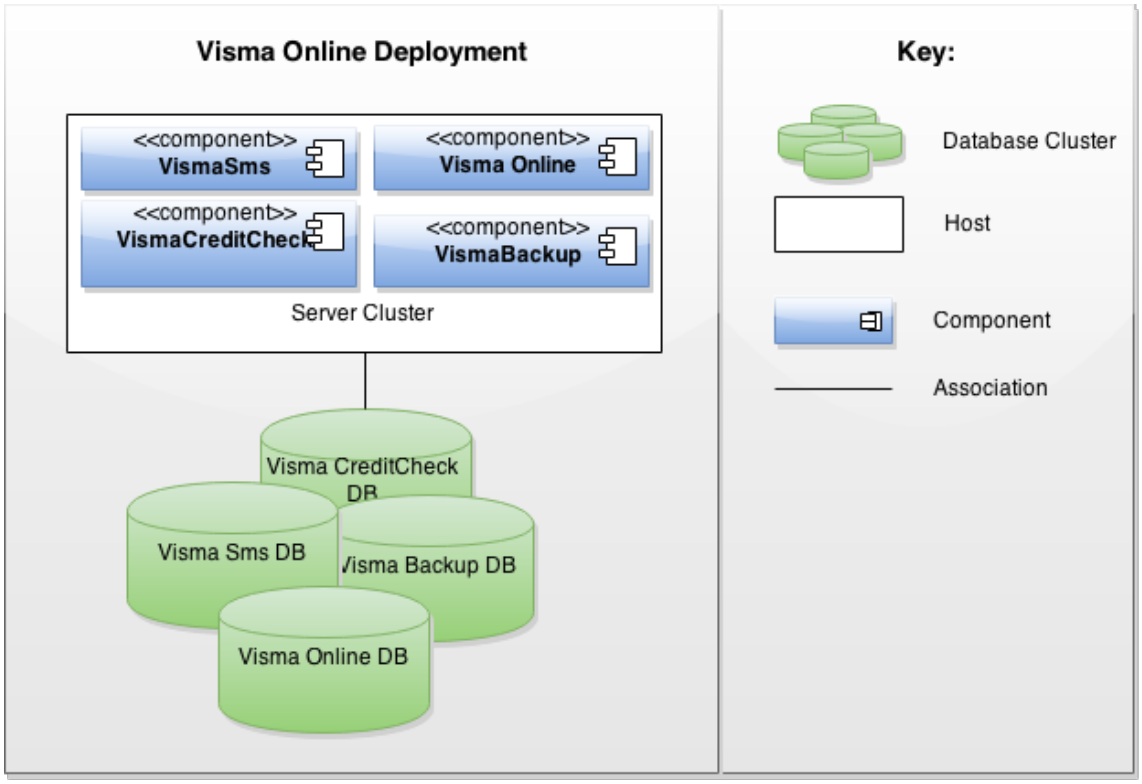


Figure 4.13: Simplified Deployment Diagram of Visma Online, Sms, CreditCheck and Backup. These components all need to be deployed on the same host, in this case a server cluster.

- Point-to-point integration [2]

As described in the theory section problems caused by these anti-patterns are among others: the full potential of SOA not being taken advantage of, the risk of a "service sprawl" [16], a high cost of change, invasive integration, limited abstraction, difficult systems to man-age, monitor, scale, secure and troubleshoot.

The Dependency Inversion Principle described in the theory chapter can also be found broken, one example is the problems described in figures 4.11 and 4.12. These diagrams show one of the classes in Visma Sms instantiating and manipulating several classes in Visma Online. Using the OO principle of dependency inversion these dependencies should not be directly towards a class, but rather an interface. Looking at the principles Visma Spcs are using for the rest of their system the communication should be through the services of Visma Online instead of direct usage of the internal classes.

4.2.5 Change Scenario Elicitation

In this section the change scenarios described in the architecture analysis method ALMA that were elicited by interviews will be presented [7]. The scenarios were split into classes to make them easier to handle. One could find an infinite number of scenarios, at the beginning of this analysis there where more scenarios than presented in this report, those who simply were not found likely to ever occur will not be presented.

Final scenario classes

1. Allocation/Deployment changes

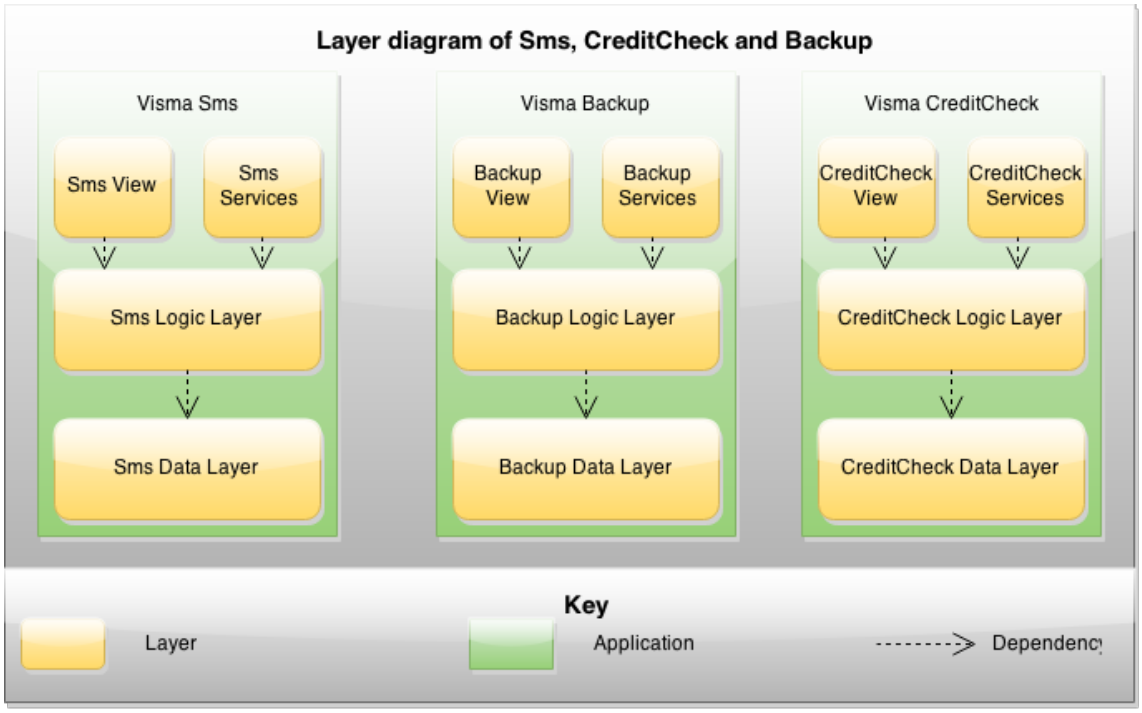


Figure 4.14: Diagram of the currect layers in Visma Sms, Backup and CreditCheck

(a) Moving Visma Sms to another server/server cluster than Visma Online
(b) Moving Visma Backup to another server/server cluster than Visma Online
(c) Moving Visma Backup to another server/server cluster than Visma Online
(d) Moving Visma CreditCheck to another server/server cluster than Visma Online
(e) Scaling physical space of a single service

2. Replacing/changing parts of Online
   (a) Changing structure of the DB
   (b) Remove deprecated API calls
   (c) Changing DB of Visma Online to non-sql DB (partly)
3. Replacing/changing module of application
   (a) Changing structure of Sms/Backup/CreditCheck DB
   (b) Changing the View of Sms
   (c) Changing the View of Backup
   (d) Changing the View of CreditCheck
   (e) Changing Sms provider of Visma Sms
   (f) Change Storage provider of Visma Backup
   (g) Change provider of Visma CreditCheck

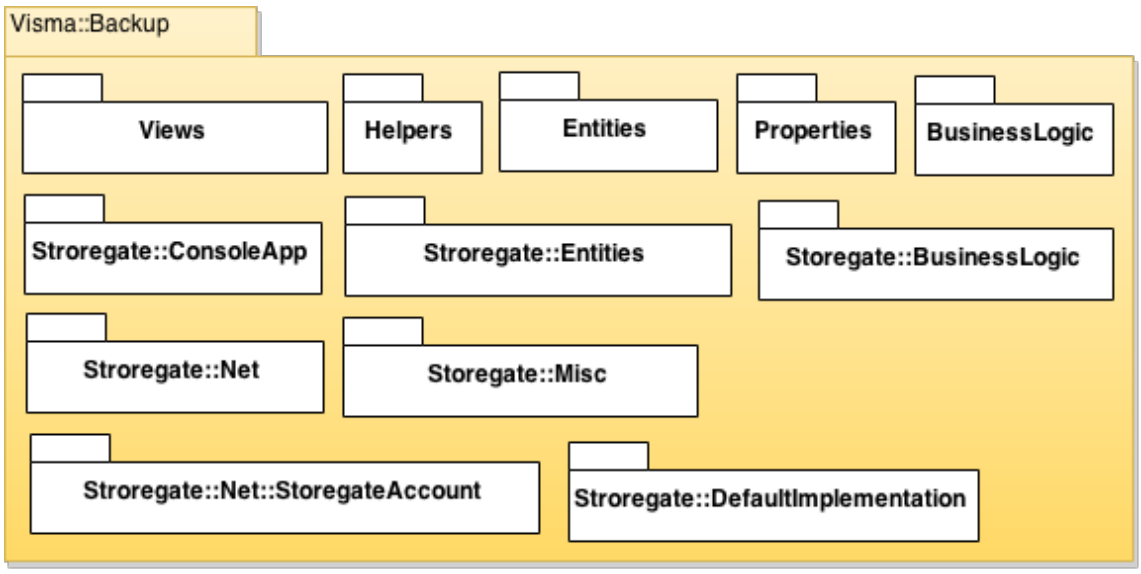


Figure 4.15: Visma Backup Package Diagram, key = UML

4.2.6 Change Scenario Evaluation

Priority and complexity was added for getting an overview of which scenarios have the top priority for the new architecture suggestion. These numbers are extracted by interviews of the architect and project manager of the Online team as well as calculated with the help of ALMAs suggested method for analysis of maintanence cost. The already extracted architecture of Visma Online was used to extract the possible lines of code affected and the ripple effect of changes. Detailed numbers will not be presented in this report.

The estimated LoC that could be changed and added per developer per day was, as suggested by Bengtsson et al. [7], calculated based on the productivity of the developers in the Online team. The estimated work per day and developer is 80 new lines of code (LoC) and 40 changed LoC. The reasoning behind this is that changing code takes more time since there are many existing modules to consider. There are ripple effects to changing code that many times are unknown to the developer when starting to make the change, while completely new modules can be made with significantly higher speed.

The scenarios that should be in focus are those that are most likely to occur and of most complexity. The results of this is as seen in the scenarios table: 1d, 1a, 1b, 1c, 2b. A more detailed analysis description of these scenarios will follow.

Moving Visma Sms to another server/server cluster than Visma Online is as the architecture looks right now regarded as too costly. If it would occur, the costs described would only apply once and then the problems would be reduced significantly.

The packages effected by this change are: Visma.Sms.Views and Visma.Sms all have references to Visma Online's Library and they need several changes to break these references, see figure 4.16 for an overview of the packages in Visma Sms. Visma Online API might also need to be extended, see figures 4.10 and 4.9 for the relationship between Online and Sms. The code that needs to be changed would be approxiamately 4 500 lines of code. If using RPC is allowed no new services would need to be added but this would mean still following older coding standards. If using services that follow REST about half of the API calls need to be added.

This leads to the conclusion that if using the older RPC techniques the cost of the change could be up to 56 developer working days according to the estimates. In this we

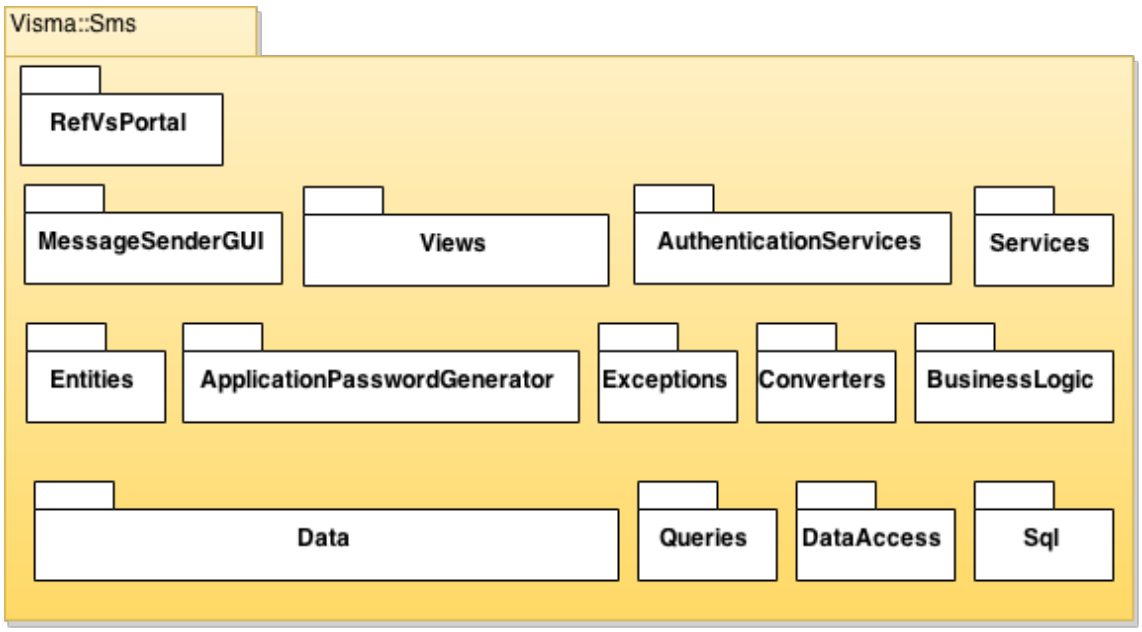


Figure 4.16: Visma Sms Package Diagram, key = UML

need to consider that there is a risk the ripple effects where underestimated or overestimated.

Moving Visma Backup to another server/server cluster than Visma Online This is as the architecture looks right now regarded as too costly. If it would occur it would happen only once and then there would be no problem moving the service again.

The packages that could be affected are Visma.Backup.Storegate, Visma.Backup.Views, see figure 4.10 and 4.9. Visma Online Services might also need to be extended. The code that might need to be changed is about 1 200 lines of code. If using the existing RPC services no new code would need to be added but if following new standards using REST about half of the API calls need to be added.

If using the older RPC techniques the cost of the change would be up to 15 developer working days. Again it needs to be taken into consideration that the ripple effects were underestimated or overestimated.

Moving Visma CreditCheck to another server/server cluster than Visma Online As the previous two scenarios the numbers presented would be a one time cost. The packages affected would be Visma.CreditCheck, Visma.CreditCheck.Services and Visma.CreditCheck.Views that currently have references to Visma Online need several changes to break these references, see figure 4.17, 4.10 and 4.9. Visma Online API might also need to be extended. The code that may be effected is approximately 6 600 lines of code. Again if allowed to use RPC no new code would need to be implemented but if using REST about half of the API calls need to be added.

If using the older RPC techniques the cost of the change would be up to 83 developer working days. Again it needs to be taken into consideration that the ripple effects were underestimated as well as overestimated.

Scaling physical space of a single application/service If it was possible to do this change without the cost being as high as currently estimated, there would be cost benefits

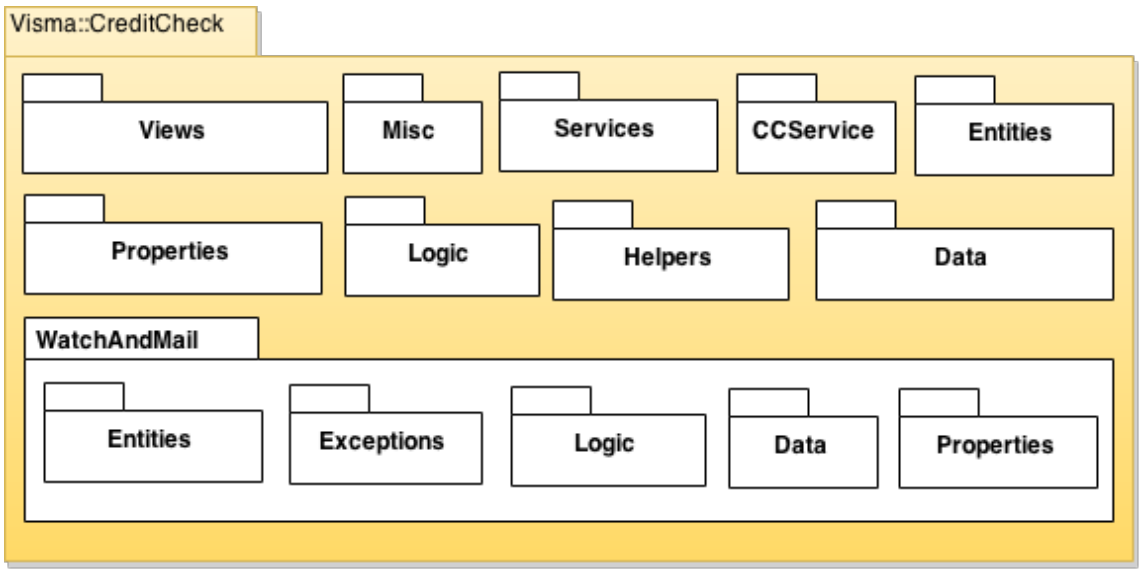


Figure 4.17: Visma CreditCheck Package Diagram, key = UML

of this occurring often.

This scenario includes moving the service and therefore affects the packages mentioned in the previous section and possibly some more. Therefore depending on which of the components space that need to be scaled the cost is different. Judging on the previous scenarios the code that needs to be changed is somewhere between 1000 - 7000 lines of code. As in the previous scenarios no new code is needed if allowed to use RPC but in the case of conforming to the principles of REST about half of the API calls need to be added.

If using the older RPC techniques the cost of the change would be up to 88 developer working days. Again it needs to be taken into consideration that the ripple effects were underestimated.

Remove deprecated API calls This is likely to occur for each release, meaning very often. This change scenario is problematic. It could be split into multiple scenarios, one for each API call. This could give a detailed view of what might need to be changed. On the other hand without access to the service consumers code bases it might be impossible to get a proper overview of what the impact exactly would be.

Let us say we change one central function: getCustomerById. This is used by Administration and likely most of the service consumers. The URI to this API call is hardcoded into Adminstration. Making a change that affects the service contract without properly warning service consumers like Administration would be devastating. On the other hand this is not something that is likely to happen due to the complications it would cause. Instead what would happen with the current principles is that the new and the old service contract are both supported until the release of the next version. Then there has been plenty of time for all the service consumers to change all the code.

Still it takes a certain amount of time to change hardcoded API calls. The packages that would be affected are the service package of the service that should be updated and every application still using the deprecated API calls. The time this would take to change is something that varies between how many service consumers use that service. Making refactoring to remove several deprecated API calls at once would obviously have a bigger effect and is more likely to happen.

ScenarioID	Priority	Complexity	Probability
1a	H	M	5
1b	H	M	5
1c	H	M	6
1d	H	H	4
2a	H	L	5
2b	H	M	4
2c	M	H	3
3a	M	L	3
3b	M	M	2
3c	M	M	2
3d	M	M	2
3e	L	M	2
3f	L	H	1
3g	L	M	2

Table 4.1: Summary of the results of analysis. The priority is on a scale of Low(L), Medium(M) and High(H) and is a combination of the complexity of the change as well as the probablility of it occuring. Complexity is on a scale of Low(L), Medium(M) and High(H) and was decided by the estimated lines of code that would be needed to changed and added for the change to be carried out. The probablity of a change occuring is described on a scale from 1-5 where 5 is almost certain. These numbers were decided with interviews of one architect and the project manager of the Online team

The reason for this being extremely likely and even necessary to occur at some point is that having deprecated API calls in the services, which might not even be used, causes an unnecessary large code base. Large amounts of code without any real use will only take space and make the code less understandable and maintainable. The APIs will be less usable and understandable for other teams. There may very well be several API calls for doing the same or similar actions. This type of duplication leads to less cohesion.

4.3 Architectural suggestion

The approach to a new architecture will be described by outlining the ideas worked with during the project and in the end presenting the final suggestion. The project does not have the scope of the entire architecture of Visma Spcs but an overview on how the new architecture fits into the current architecture and how it should fit in the future.

Modifiability has been the key in the investigation. Visma Spcs system architecture is a constantly changing and growing creature that never stays the same for long. Scalability is extremely important for a system that is expanding as fast as Visma Spcs. Loose coupling is the main principle of making the SA modifiable. As seen in the analysis of the previous architecture there are some scenarios that are more likely to occur than others, and some scenarios occur very often even. There are also those that have a higher cost once they occur. The focus will be to reduce the cost of implementing these scenarios.

The fianl suggestion builds on the principle of low coupling and high cohesion. The cohesion is in this case not a large problem. Visma Spcs system is divided neatly into applications silos and services each reflecting business goals. The first idea was to build on these components but decoupling them from Visma Online, see figure 4.19. Seeing

how this caused some problems it was instead considered keeping the services of Sms, Backup and CreditCheck within the Online service inventory and decoupling the views, see figure 4.25. In this way we would miss out on some essential requirements which lead to the solution of using decoupled services and decoupled views together with the pattern API gateway, see figure 4.26.

4.3.1 Moving application silos away from the plattform

Moving the applications away from the plattform but keeping them as applications silos, as they currently are was one very intuitive solution to avoid reinventing the wheel. The anti-pattern of keeping application silos when moving over to a SOA architecture has been described as well as the reasons for not doing this. Even more troublesome in the current architecture is that these application silos live within Visma Online, which should serve as the main service inventory.

One of the important changes of the architecture should be to move any views reciting in the service inventory, because they do not contribute to any reusable logic which is the main goal of the service inventory. One solution would be to move Visma Sms, Backup and CreditCheck to be loosely coupled with Visma Online. The way to do this will be to make these applications communicate to Visma Online through the services provided with the same rules as other legacy applications, see figures 4.18 and 4.19.

Disadvantages of this approach includes that in order to gain the loose coupling the master page from Visma Online can no longer be directly referred to. Making the master page as a public resource is somewhat more complicated than the majority of the logic in Visma Online.

Advantages of this approach

- Visma Online is a clean core.
- Visma Online, Sms, Backup and CreditCheck can more easily be moved to different hosts.
- Visma Online, Sms, Backup and CreditCheck can more easily be scaled independently.

MasterPage is currently not shared between decoupled applications. In order to completely decouple Online there are two options: copy the master page layout and put it in the three applications or make it a shared resource. The first option would mean low cohesion when it comes to the duplication of code. Changes in the MasterPage would mean changes in multiple places, since the master pages are copies. Which his why the second option is to be preferred.

What remains is a service oriented platform with legacy application silos. This architecture looks a lot like the hub pattern in EAI, but when taking a closer look at one service consumer, in this case at how the desktop application Administration is going to use these services, it starts to look more like a point to point integration. Compare figure 4.20 and figure 2.3a.

Other anti-patterns discussed was the ones about not letting the application silos remain, again see figure 4.20, and not treating SOA as EAI 2.0 [16]. Using this approach the change that brings the advantages we are seeking for also gives some anti-patterns

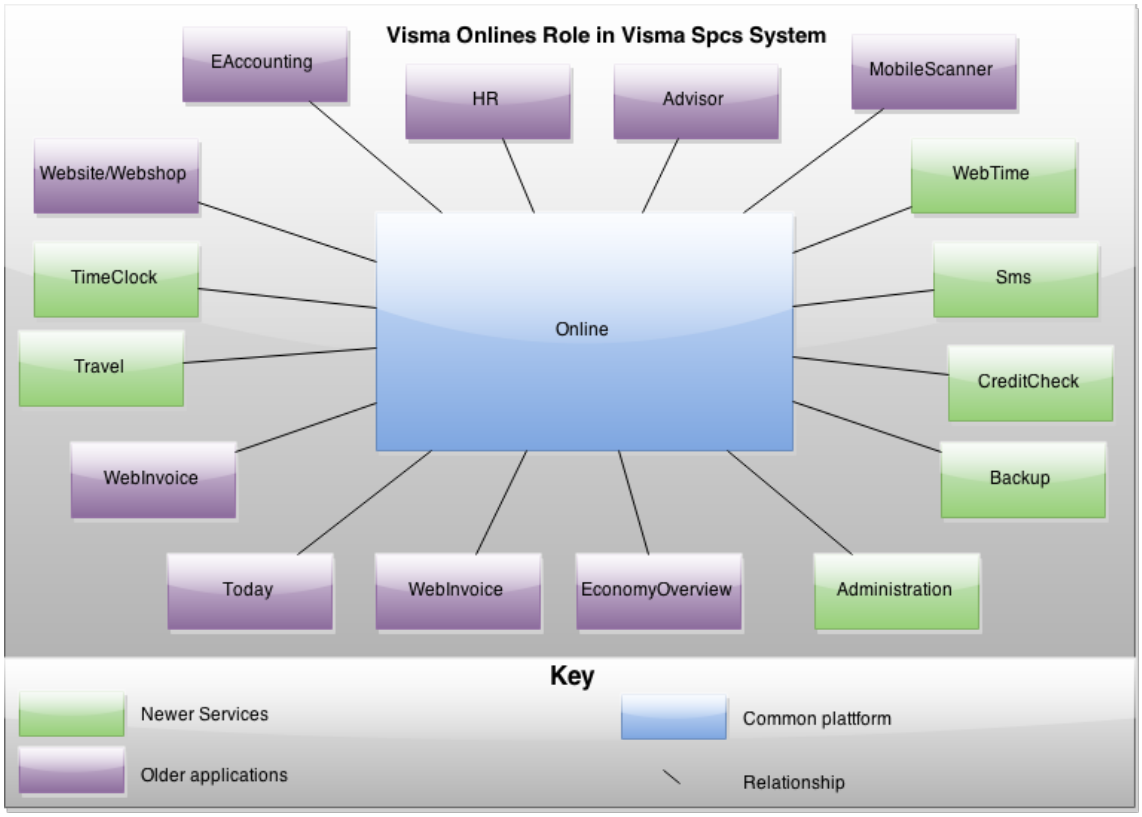


Figure 4.18: New plattfrom overview

each with their own problems. Also the principle in SOA for loosely coupled services is broken when keeping the views coupled to the services. This brings us to another suggestion.

4.3.2 Loosely coupled services

Loosely coupled services could be achieved since layered applications like Visma Sms, Backup and CreditCheck, see figure 4.14, are not that far from being a backend that serves as a service provider and a frontend that is a service consumer. What is missing is that the view communicate through services instead of the interfaces currently used. Sms, Backup and CreditCheck already have services for other applications or services to communicate through, see figure 4.22.

These applications are not far from already following the idea of microservices. As the other services in Visma Spcs system, for example EAccounting, they do not necessarily follow the idea of being micro, but they are loosely coupled services with high cohesion divided depending on the business goals. Sms, CreditCheck and Backup all fit into these prerequisites and makes an excellent addition to Vismas Spcs newer take on SOA.

We can also look on the system from a service point of view. For the sake of simplicity and giving a clear image of the idea the legacy applications will be left out from this view. From this view the system constructed with multiple services and frontends. Visma Online is one large service and Sms, CreditCheck and Backup are smaller services. Multiple views (legacy desktop applications like Visma Administration as well) communicate to these services via different APIs with different contracts.

Each service can be built separately and deployed on different hosts. Each service has its own database. With this approach the problem with the point to point integration

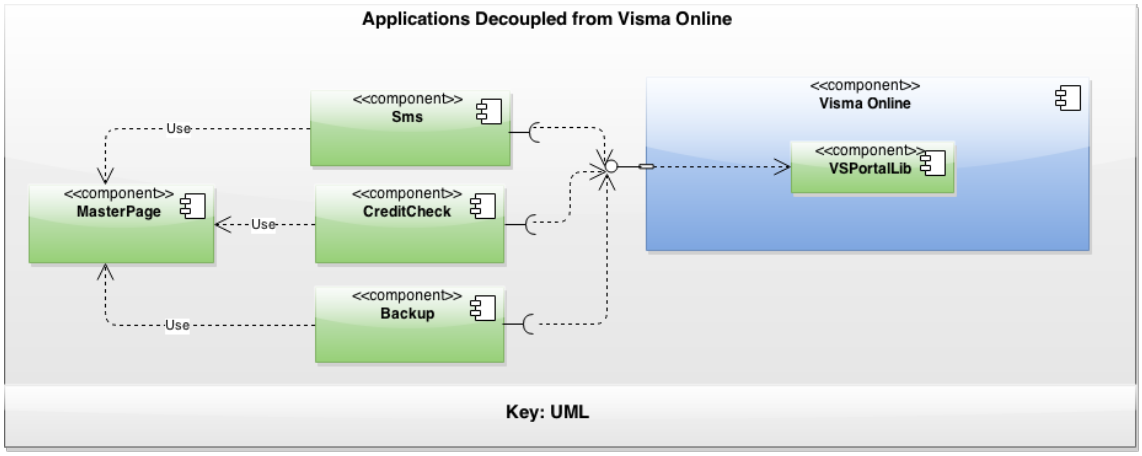


Figure 4.19: Sms, CreditCheck and Backup remaining as applications but loosely coupled from Online.

remains, even more so when the views of the applications are seen as own components. When it comes to the separate builds for the services this is one approach to support CI, see figure 2.4. According to Newman this approach has the advantage of only having to run the tests necessary for that build, the fact that there will be one artifact to deploy and that the ownership of the code is clearer (if one sees this as an advantage) [3]. As well as having one separate build per service as an approach to support CI, similarily there can be one build pipeline per service to support CD [3].

Again comparing figure 2.3a and the new suggestion ahowed in figure 4.23 there is tight coupling when it comes to the knowledge of the service consumer, in this case seeing the former view layers of the applications and also the administration desktop application as service consumer this is quite some extra connections needed.

4.3.3 Splitting or not splitting Online

Splitting the *monolithic* service inventory, in this case Visma Online, is one part of the idea of microservices [3]. Visma Online already has separation between different services just not to that extent that they are completely loosely coupled and communicate via service calls.

The main trade-off of splitting the services is that communication only through service calls reduce performance. Another is if the databases are also completely separated syncing between databases might become a problem in specific cases.

So should the services instead remain in the bounds of the Service Inventory Visma Online? Making it invisible to the service consumers and in that way removing this point to point pattern seen in the previous architectural suggestions? This means still treating the views as service consumers instead of keeping the views tightly coupled with the service backends, see figure 4.25. This would mean keeping the backend of the application coupled to packages in Visma Online. Problematic coupling like seen in 4.11 and 4.12 would still need to be handled but dependencies to interfaces instead of classes is a solution to this using the OO approach instead of SOA.

Advantages of smaller loosely coupled services are among others the risks reduced from changing or even replacing one service provider or service consumer, as long as the service contract does not change, only restricted to the boundary of that service. This

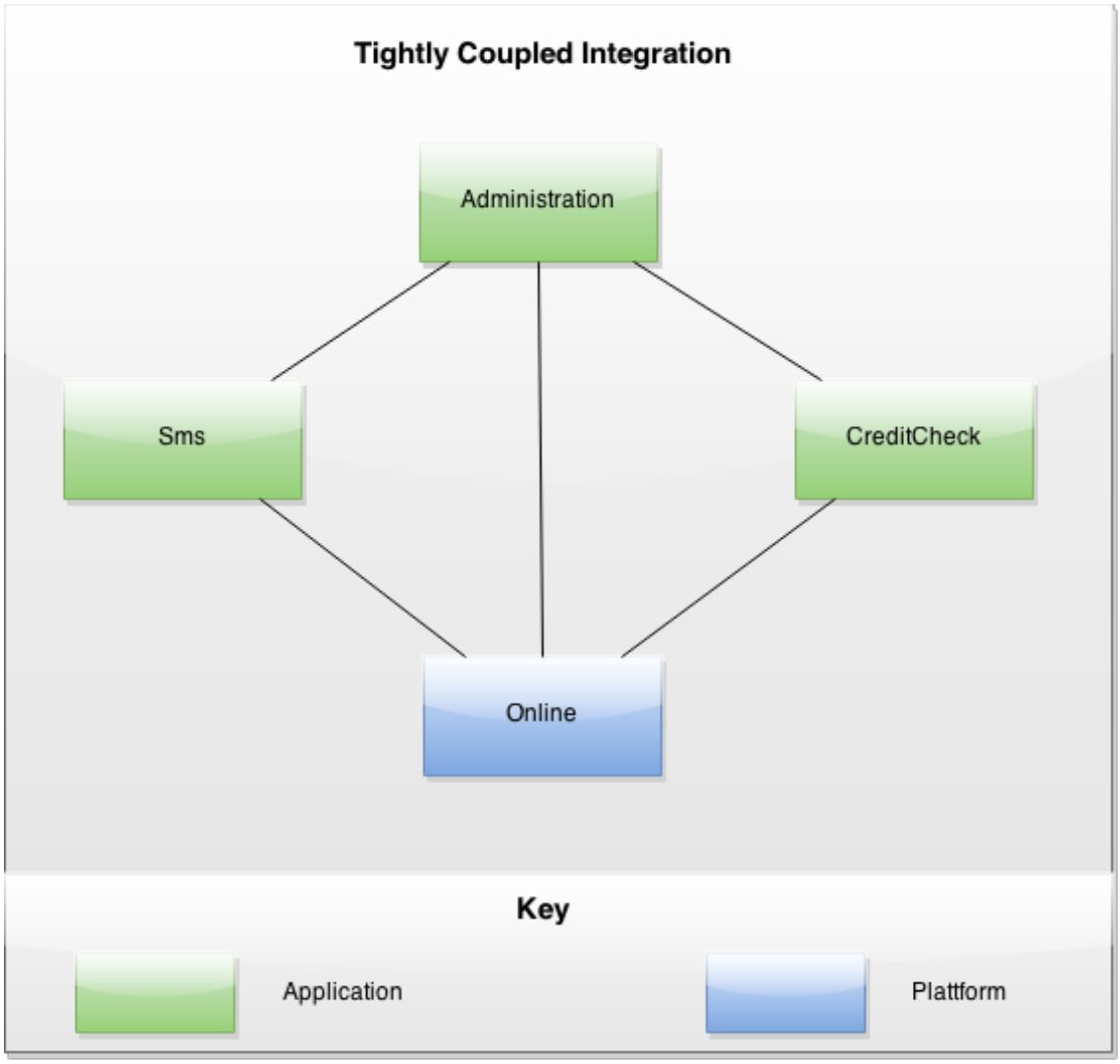


Figure 4.20: The integration between applications looks like the beginning of a point-to-point anti-pattern.

makes them replaceable. The fact that the knowledge of the service consumers does not need to go beyond the service contracts enables smaller teams working independently, even on different locations.

The smaller the service the more the benefits from interdependence increase. The ideas of smaller services can solve the problems caused by code bases that grows when adding features. These large code bases can make it difficult to see where changes needs to be made, even when striving for clear modularity. Fixing bugs or implementations is difficult to do. The idea of high cohesion is an important concept within microservices and as the SOA principle the boundaries should be business-driven. With increased cohesion it should make it obvious where changes in code needs to be made.

If each service can be deployed independently of other services it will be easier and less costly to deploy new versions of services, or even replace them completely. This also makes the long term commitment to technology easier to get rid of.

The scalability also improves. The approach enables organizing the development effort around multiple teams responsible for a single service that can develop, deploy and scale their service independently of other teams.

Fault isolation can also improve. A memory leak in one service will not affect other

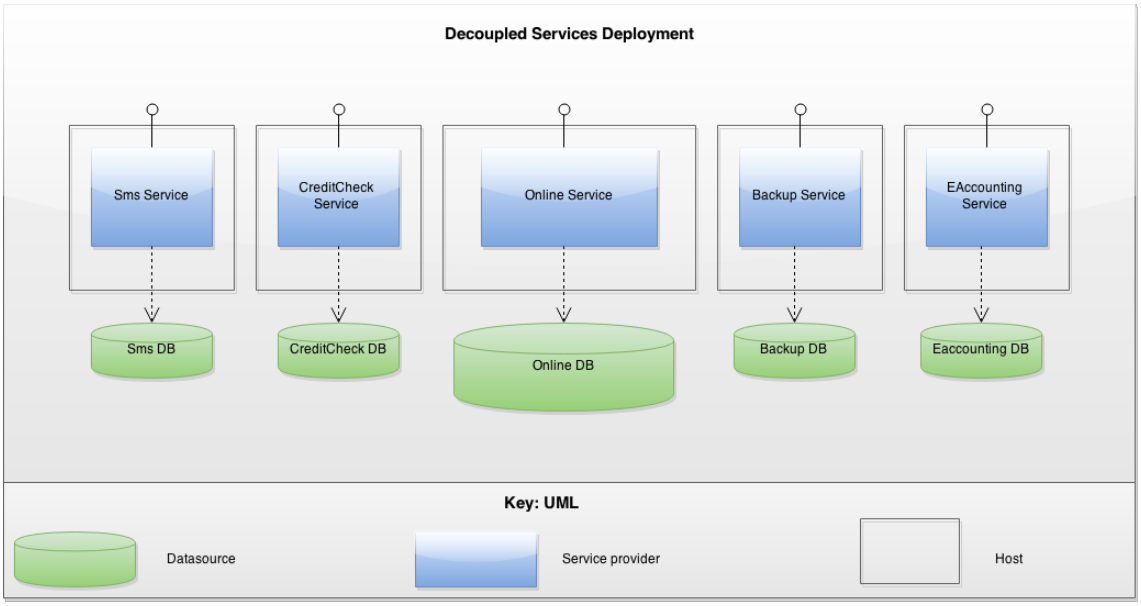


Figure 4.21: View on the idea of having Sms, CreditCheck and Backup as services on separate hosts.

services in the same manner as in a monolithic architecture where that can bring down the entire structure[3].

Disadvantages with smaller services When the services become smaller complexity from having more moving parts emerges [3].

- The complexity of testing increases.
- The communication between services must be implemented.
- Deployment becomes more complex.
- Increased memory consumtion.

Can we have the best of the two worlds? Seeing the problems about choosing a single point of entry or loose coupling between the services, both with advantages and disadvantages made me search for some option that could give the best from the two worlds. The services of Visma Online should be able to be split into smaller autonomous services, having the views as loosely coupled service consumer and still avoid the point-to-point pattern. There could be a layer between the services and the service consumers, like the broker pattern described in the theory section. In SOA this pattern can be used with an API gateway as a broker, see figure 4.26 [22]. This way the idea of having autonomous loosely coupled services that can be deployed on different hosts, see figure 4.21, is still possible. This makes it easy to scale the physical space for a single service. The Views are still loosely coupled from the backends of the services and instead communicating through the services, see figure 4.22, but there is the API gateway layer in between hiding which service is being called. The services Sms, Backup and CreditCheck are loosely coupled from the other services of Visma Online, see figure 4.19, making Online a smaller code base. Also the troublesome coupling seen in figure 4.11 and 4.12 will no longer be a problem when communicating through the API gateway, see figure 4.27.

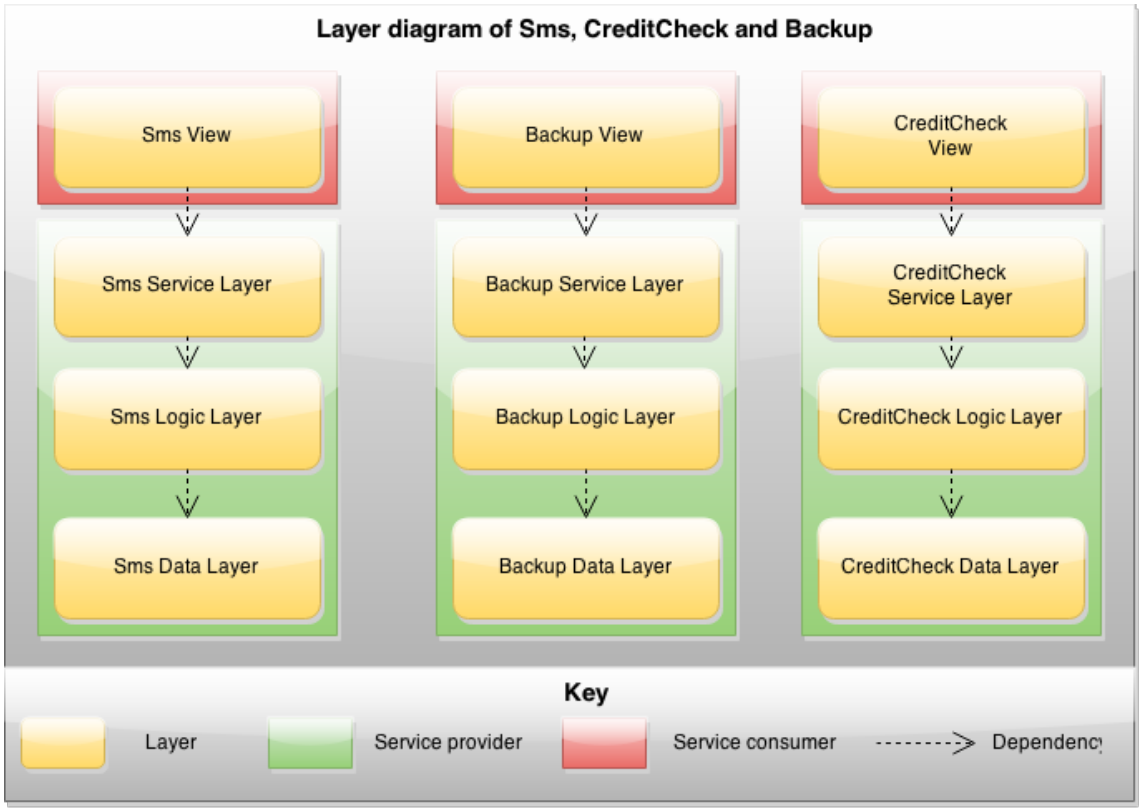


Figure 4.22: The Views as service consumers and backend and service providers.

Using an API gateway there will be a single point of entry between the service consumers and the services. There will be no knowledge in the service consumers as of which service is being called from the API gateway when a service consumer makes a request. Replacement of one service or change of services is handled in the API gateway making these changes easier to make. We can almost get to that point where connecting a new component is just a matter of hooking it in at one place, then it is ready to be used.

Finding a new improved way of handling the versioning of services is also made easier by the API gateway since this logic can be handled at a single point. There can be several versions of an API for a time, both being used but the API gateway handling what amount of requests going through each. This way a new version can be tested on the fly. Once the older version is to put out of use the new version should be the primary version and the older version can safely be removed. This is sometimes referred to as a canary-release [3]. This is one way to keep the code base from being filled with out of date code that is no longer used.

The trade-offs are the same as described in the theory section about the broker pattern. The main concerns are that the API gateway can become a bottleneck of performance and a security risk if not implemented carefully. There are however API management solutions and gateways out of the box from third parties [23]. Even if these solutions may cost money, the upfront cost and implementation of a complex API gateway is removed.

4.4 Comparison of Architectures

Sms, CreditCheck and Backup are all applications that are stable and rarely changed. The layered architecture of the applications causes no direct problems and these change

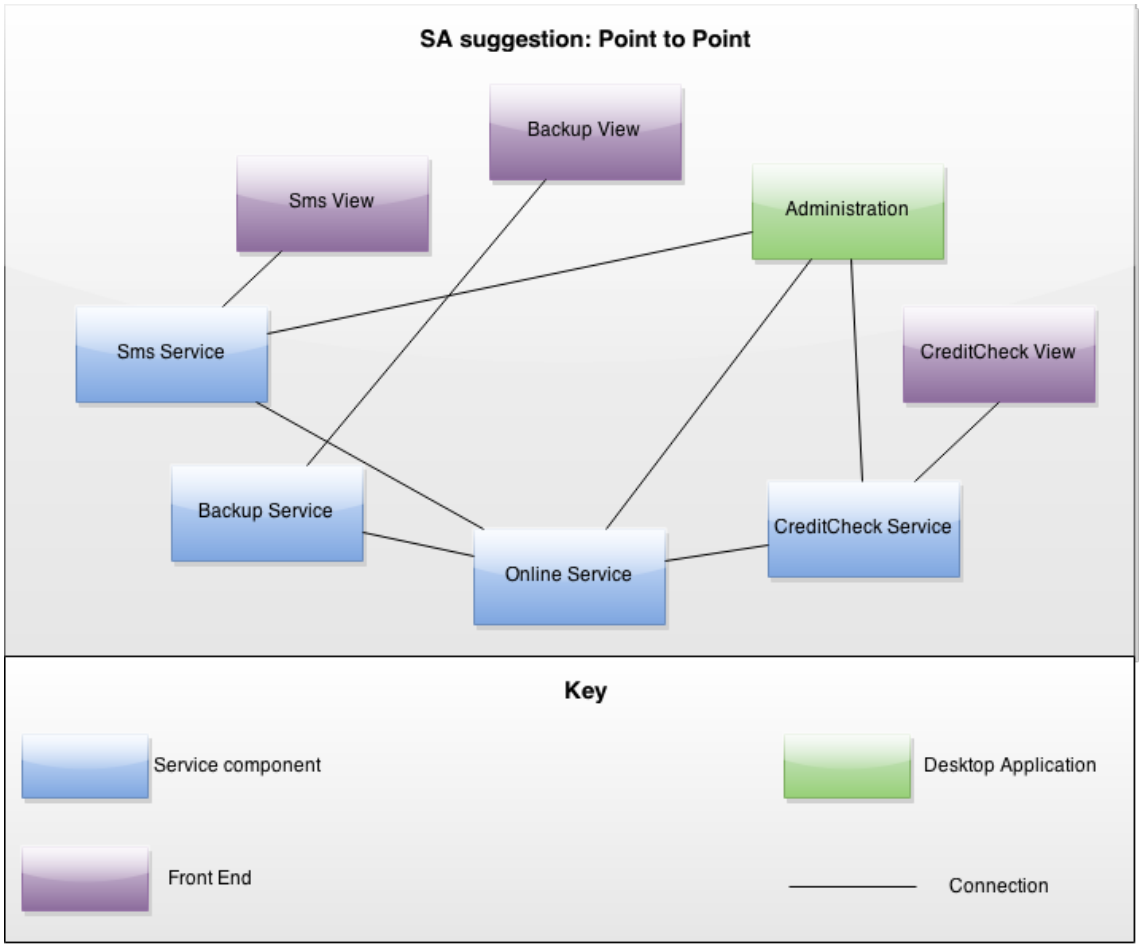


Figure 4.23: This figure shows a beginning of the point-to-point anti-pattern.

scenarios got rather low priority when doing the analysis.

Moving Visma Sms/CredidCheck or Backup to another host than Visma Online This is as the architecture looks right now regarded as too costly. If it would happen it would only be a one time cost, upcoming changes most of the difficulty with moving would have been handled. The code that needs to be changed in the current architecture was estimated to 1200-6600 lines of code which is approximately 56-83 developer days, if following the old service principles and using existing service contracts. With the suggestion there would need to be no lines of code added or changed.

If Scaling physical space of a single service was possible without changing so much there would be cost benefits of this occurring often. This scenario would include moving the service and therefore depending on which of the components that needs scaling this scenario would mean a change of around 1000-7000 lines of code, which at most is 88 developer days.

The option to scaling the space of a single service when necessary is to always scale the services together, as is done now. This brings in to account a hardware cost instead. The flexibility of not being able to move one of the services to a cloud host like Azure where there is a possibility of reducing cost when there are fewer users should also be taken into account when deciding if this is a cost efficient change at all. With the new architecture this change scenario would not require any or insignificantly little change in code. Making the suggested architecture cost efficient if this change needs to occur, which

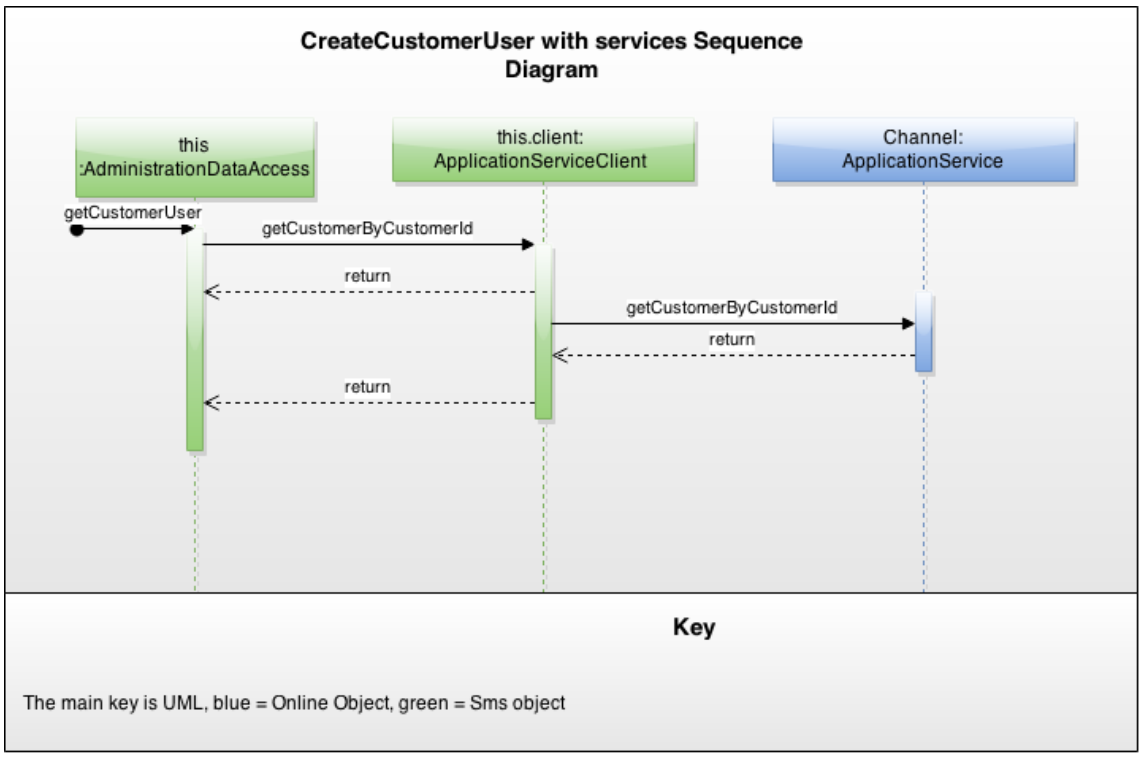


Figure 4.24: Sequence diagram depicting using the same technique Administration is for communicating with the services of Visma Online. This means not adding new services, but Sms will still have strong knowledge about the services in Visma Online.

is likely.

In the current architecture removing deprecated API calls would mean changing the service consumers calls to the services. Using versioning this can be spread out over a longer period of time. However the changes still need to be made and depending on how many service consumers are using the service the cost could be very high.

With the API gateway the change will be isolated to one component. Once the change is done in the API gateway the old version of the API call can safely be removed saving time and space. With the idea of canary releases using versioning the API gateway may also be designed to test new API calls by handing over only a few amount of requests to the new version, while the majority of the requests still lead to the old version. This can work as a test of a new version reducing the risk of continuous integration and continuous delivery. However tha main advantage of the API gateway in this context is the fact that a new version of a service does not mean changes have to be made in several places, since the service consumer contacts tha API gateway which handles the logic for the requests.

4.5 Analysis

During this case study I have gone through several architectural patterns and strategies that have been used by Visma Spcs and that could be used to solve the current problems.

4.5.1 Online as a plattform

This is the first intention of Visma Online. The applications are however built at the time used classes from Visma Online freely, causing this tight coupling. The advantages of this approach was mainly that it was not a complex solution, making it very easy for the

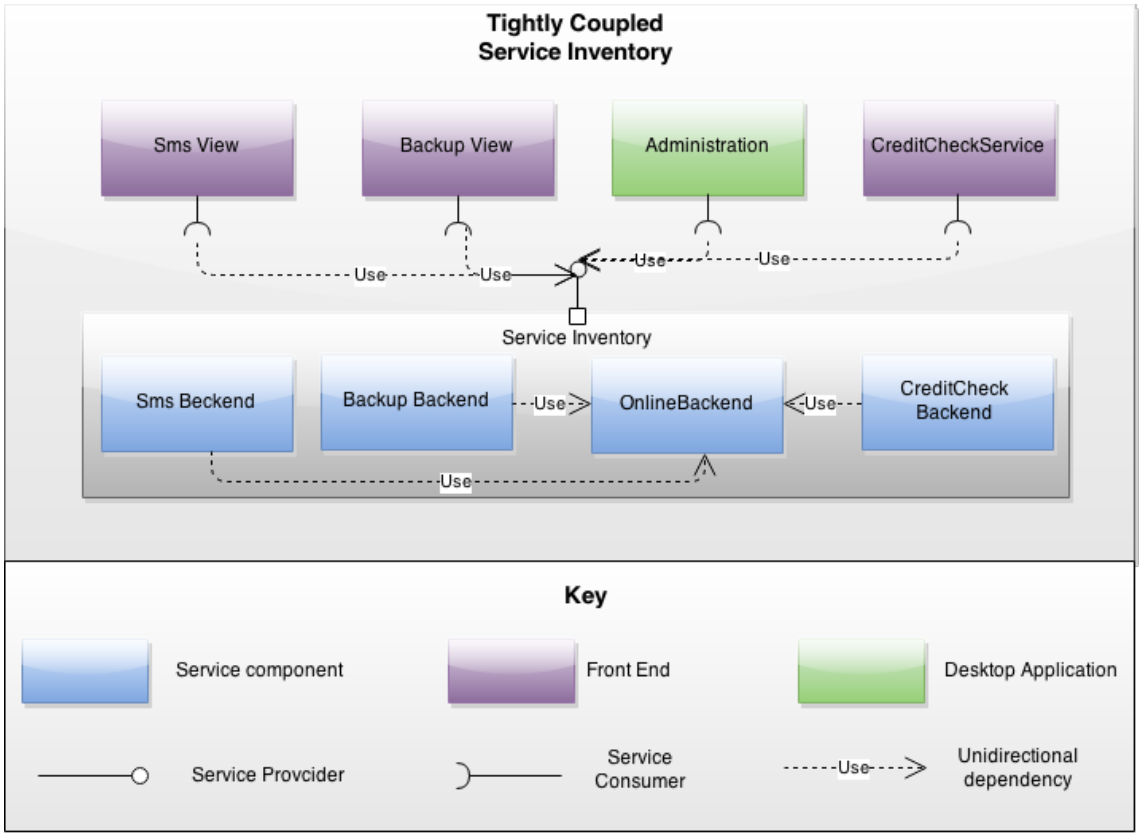


Figure 4.25: The services remaining in Visma Online with a single interface towards the service consumers

developers to understand and use Visma Online's functionality. The architects realised rather quickly that this tight coupling caused the project to grow large, the build artifacts depended on each other, the tests got slower and the deployment ended up being on the same host. Obviously this approach could not go on but some communication model needed to be established in order to avoid these problems. Since there has after the decision of using services for communication been an explosion of services and applications one can with hindsight say that it was a very reasonable decision to move away from the previous strategy.

Having Visma Online as a common service inventory where Sms, CreditCheck and Backup may be some of the services provided. This would require some standardization to be able to be pulled off but would end up in a rather easy SOA model. As described the inventory code would eventually grow rather large, the problems with scalability and the slow tests would remain. Which were some of the most important requirements to improve which made this approach not a very good one.

4.5.2 Newer services

Loose coupling between backends and frontends as well as between applications makes a change in either frontend of backend more isolated. The ripple effects of having to perform a change is therefore less. Replacing the backend or frontend of any application will not affect the rest of the system as long as the service contract is still fulfilled. This approach also opens up the options of deploying each component at different hosts, which with the right cloud host can turn out to be cost efficient compared to hardware cost. The communication in between services can become chatty and affect performance. It goes

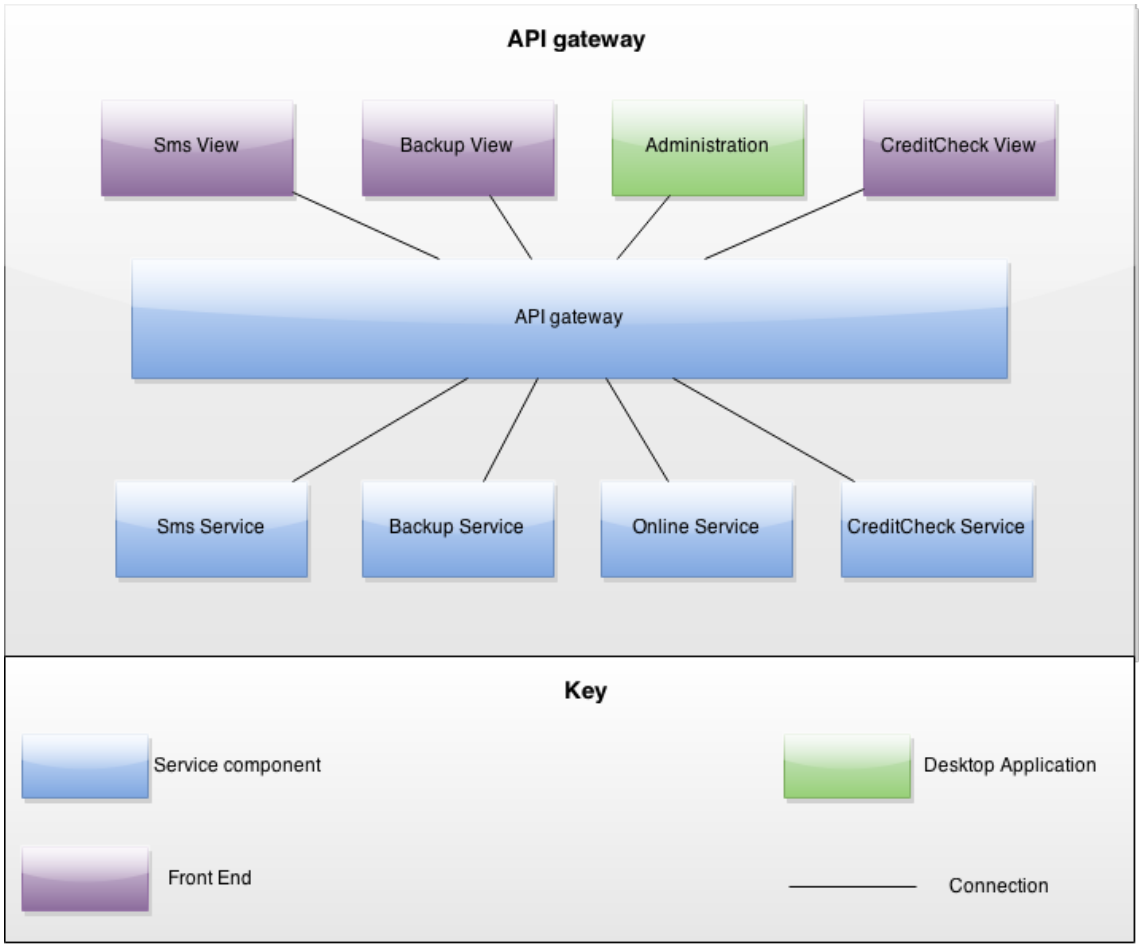


Figure 4.26: Having an API gateway layer to work around the point-to-point pattern arising from having smaller autonomous services.

without saying that using this from-case to case approach to services without documenting will be confusing and time consuming. Changing the service contract of one service may affect several other applications. One solution to this is versioning of the services allowing applications to change the version when there is enough time.

4.5.3 Using an API gateway

An API gateway could standardize the way service consumer communicate through the services, weather this being the Sms service, CreditCheck service or Online services. This approach could mean that each service can be deployed completely separately, implemented in different languages, following different standards and still the service consumers can easily use all the services. Also in the case of versioning, the possibility of not letting this even noticeable by the service consumers increases. This simply leads to added complexity by having multiple small services not noticeable for the consumers, which makes it easier for a developer on the consumer side to start using the APIs and therefore saves time when it comes to learning. The main disadvantages of this approach is the API gateway becoming rather complex to design, there are several risks involved if it is designed incorrectly such as the risk of and API gateway becoming a security risk as well as a performance bottleneck. There are third party API management tools to use that include an API gateway, even if these solutions may cost money it does avoid you to have to reinvent the wheel, which also has a substantial cost. As seen in the comparison,

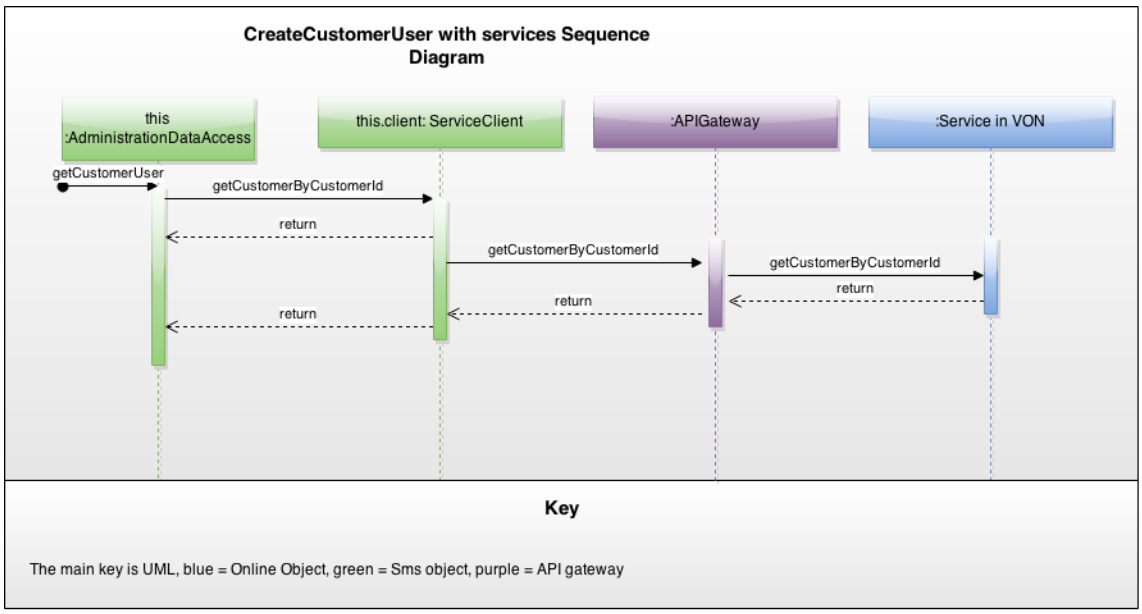


Figure 4.27: Sequence diagram depicting using the API gateway as a layer between the Online services. The service consumer, Sms now does not need to know which services are being called from the API gateway.

strictly speaking in lines of code the suggested architecture would be time saving in those scenarios. However there is also the cost of change to consider. When it comes to the scenarios of deployment and allocation the change of the architecture would probably cost more than the one time cost of performing the scenario in the current architecture.

The numbers presented in the analysis of the current architecture however assume that using services that already exist and that they are performed by someone familiar with all these different technologies and principles that knows in what situation to use which. The complexity of the system would therefore remain the same.

The advantage is however that once a component is decoupled, by having a own built artifact, this component could later be deployed and scaled without any real effort at all, so this could be done without considering the cost. If these deployment changes and scaling of a single service is to happen frequently the decoupling would have had an advantage.

4.5.4 Remodelling the system

Systems needs to be remodelled at times of change. Following different sets of principles within the same project is as set up for lack of understanding and principles being broken. Remodelling everything at once means a big upfront cost. One solution to both these problems is that new code always follow new principles, causing the system evolve slowly. As well as performing software recovery to avoid the risk of troublesome software erosion.

Should there be more decoupling in the future? Or is it enough to decouple these services? Modifiability is the key in this question as well. Where the lines are drawn depends on what we can see that the future holds, it depends on how much the system will grow. Visma Online is already an enormous module, splitting it might not be such a bad thing. But changes in architecture takes time, which costs money. Following one architecture makes things easier to understand. But in the case of Visma Spcs there is already several patterns being used and different principles followed, mostly depending on when the implementation was done. For the sake of not having to rewrite the older applications

altogether it is very likely that these will remain in their state of being application silos. However Sms, CreditCheck and Backup are already layered and not as difficult to change as the older not-layered applications. This is really a question of dealing with "if it's not broke don't try to fix it" or doing a not so big change that makes the architecture more coherent and easier to understand.

Some benefits were made by the change of architecture. This was a small change but it took time. The main time spent was on the SA recovery, trying to understand what the SA looks like and what views are relevant. If the SA documentation would have been up to date this would not have been a big project - approximately 1-7 kLoC was to be affected by the changes and this is at most 88 full developer days.

This developing time is however a one time cost, the next time changes needs to be made the architecture is more flexible and this means the changes will take less time. After many small changes put together the cost of changing the system have been reduced, which will eventually be a time-saving architecture approach. The risks that are always faced in such a large project is the risk of the code in the project being unmanageably large.

5 Discussion

"—It is clear, said he, that things cannot be otherwise than they are, for since everything is made to serve an end, everything necessarily serves the best end." - Voltaire

If it's not broke don't try to fix it, is a common way of thinking and I ended up thinking about taking it during this project. Perhaps it is as Pangloss believed that we already live in the best of all worlds and everything serves it purpose perfectly, most likely this is not the case and if we keep that mindset we are very likely will be proven wrong. Even though there might not be an Eldorado for the world of computer science it is no reason to just be content with what we have without atleast looking into the possibilities of improvement instead of assuming what we have is already perfect. I ended up think that if something is on a risky path towards becoming broke you should fix it before it is to late. Changing something that is not broken means it can still be functional while performing the changes. Trying to change something that already is broken can be stressful and might lead to last minute solutions that will not hold in the end and actually end up having to be fixed again sometime soon.

5.1 Reflection about Visma Online

Visma Online was first developed as a common knowledge base or service inventory. Over time more and more functionality was added and Visma Online started becoming a large code base which showed difficult to modify, test and deploy. The architects working at Spcs saw this as a problem and came up with the solution of decoupling newer functionality from Visma Online. Even newer developed functionality has decoupled views from services as well with the goal of following the principles of SOA better. But the legacy remained from before these decisions were made. Part of this legacy is Sms, CreditCheck and Backup where the views are coupled to the underlying logic of the services and the logic as well as the views are coupled to the logic of Onlines services. Remodelling this already existing code has been on the to-do list for a long time. Since remodelling is not a bug nor new requirement the priority for this task has been low. Instead new code being implemented is supposed to follow new principles. Since in spite of how much design that is done and principles are set the final architecture is decided during the implementation and the principles and patterns that are supposed to be followed is not the ultimate truth about the real architecture. The truth is that code developed should follow the current principles but it is up to each individual developer to follow them or not.

5.2 Reflection about architecture erosion

The attributes given for software by Brooks complexity, conformity, changeability and invisibility still holds solid [13]. These attributes are in the core to why software architecture in the conceptional stage looks different than what is later implemented. Complexity leads to communication issues between architects and developers, amongst others. Lack of understanding the architecture or the system leads to accidental missteps from the original design. When on top of this the system is in constant change, the requirements are in constant change and the world around us is in constant change the architecture will need to change. If all these changes are not thought through as of how to fit into the original design it is obvious why reality and architecture is diverging.

5.3 Reflection about the architecture suggestions

When there is only a few of these parts of a project with some undesirable attributes it will not be a problem and can wait until the next time these parts will need to be reworked. Keep making the newer projects follow the standards to slowly move towards the goal is in my opinion a very reasonable approach for a company that is business driven. The end goal is in this case to have a service oriented architecture which is business driven and loosely coupled enough that the entire service inventory is not necessarily deployed in one place, since this is what is working best with the business goals.

Unfortunately with an iterative approach these design decisions made in the past live on for a long time. It might take a long time for making changes in stable applications, so the old architecture can not simple be erased over night as new smarter design decisions come along. This is something that is true in most cases for a large and complex system. The choices are between remodelling everything to a new architecture as soon as you discover the problems or doing it step by step. The step by step approach does not hinder new business goals from being achieved while correcting mistakes from the past. This said keeping the goal in mind of highly reusable and flexible services is important. This is why some of my suggestions for architectures would only take the architecture one step forward but two steps back. Treating the parts of the project I have been looking at as legacy systems is not appropriate because we can say almost for certain that they will change some day, at least the views. When we are already considering remodelling these part we might as well do it well, or as well as possible from the cards we are dealt.

The suggestion I eventually ended up giving fits well with the new part of Visma Spcs software architecture, for example with EAccounting. The API gateway is something added because of the fact that more services that are loosely coupled gives more endpoints and this adds complexity. The more you do this the more we add to the complexity for the service consumers. Not wanting to add even more trouble to the service consumers was my main goal with suggesting this API gateway. As it turns out this was not a completely new idea or something that has not been discussed before in the project. Originality seems hard to find, but I will take this as a confirmation that my idea is something that fits into the goal of the company and actually is executable in the future since simply decoupling the services will bring trade-offs as well as positive effects, which if carefully designed the API gateway can avoid with fewer trade-offs made.

What seems to not be avoidable is the complexity of the system. This is a large company with many services offered, it is bound to get complex at one point or another. Realising where improvements can be made to make this less complex is important for the fast delivery of new products or improved products to the users. I do not want to claim that my solution to the problems I encountered while decoupling Visma Online is some one-size fits all solution to architecture. I strongly believe that a software architecture must match the business architecture in order to be successful. This is one suggested solution to a specific problem I noticed.

5.4 Reflection about problem solving and results

I found that the most intuitive solution I first came up with for decoupling the services actually ended up in something looking like the point-to-point anti-pattern. Adding more services to this would have created an integration mesh with each component having too much knowledge of the other components. If not haven thought one step further than only loose coupling of the services, without taking the service consumers like administration into consideration that architecture could actually have caused more trouble than it was

worth for the users of the services, ending up in not having accomplished anything but adding more complexity to an already large and complex system. This is how easy it is to create anti-patterns when lacking enough previous knowledge and just goes to show how important the work of software architecture can actually be, also how important it can be to think more than once before making any decision.

5.5 Method reflection

The most time spent on this project was on architecture recovery. This was not something I anticipated, in fact I would never have imagined how difficult it can be to get a good overview of such a large system architecture. I ended up not only having to have a grasp of the software architecture but the systems architecture and the business architecture as well to even feel like I could say anything about what might be the best way to decouple these services. Trying to figure out where dependencies where and what ripple effect might come from the changes needing to be made was not easy either. If I got a chance to do this a second time around I would have chosen to follow some iterative approach like Focus, building up the architecture while coding seems after my work like a better approach. My intuitive approach to this was to get all the information I could about the SA at the beginning, then analysing where the problems were and then trying to find suggestions as of how to make those problems go away. While reading about patterns to solve the problems I actually got a better image of what was going on right in the current architecture as well so this ended up being an iterative work even if that was not the initial plan. Another point to make is that this goes to show how important it is to have architectural documented. I have even more respect for the importance of documentation. Documentation also takes time, but during my work the documentation of the architecture was not even half of the work of trying to figure out what was going on.

6 Conclusion

"The only way to make sense out of change is to plunge into it, move with it, and join the dance" - Alan Watts

In our line of business things changes in a rapid rate. When a company starts developing their software system there is no knowing what the future may hold. The world might change to the degree that nothing of what the users want today is desirable in a couple of years. Users may surprise us in their changing requirements and trends change almost over night.

When Visma started building their own software it was not a large application and there were not that many users. Now Visma Spcs is the largest in their line of business in Sweden. Their system consists of so many applications, desktop applications, web applications as well as SaaS solutions. During the time I worked on analysing their architecture I asked myself: "Who knew this was where they would be?". Asking the most senior developers, it seems no one knew, no one could have known what new technologies would come, or for that matter what new technologies will come. For a B2B company like Visma Spcs with several competitors it is crucial to keep up with the newest and most desirable technologies for the users. Which leads me to the architecture, the only thing an architecture in this fast changing world really can prepare for is change. The only thing we can ever be certain of in our world is that nothing is certain, this applies of course as a philosophy in many areas but within modern software development in particular. In my opinion the software architecture has the responsibility of preparing for these changes that may or may not occure, changes that we can not even dream of yet should still be possible to make with minimum effort. This is no easy task but loose coupling and high cohesion does take us a long way.

6.1 Is there a better architectural strategy for Visma Online services to decrease the degree of coupling?

There are multiple patterns and strategies available to decrese the coupling and get a more cost efficient software architecture that is prepared for future change. One strategy that fits in the B2B world which Visma Spcs recites in is SOA. This is something that already was widely spread across Visma Spcs system. However the still used legacy applications Visma Sms, CreditCheck and Backup is still being tightly coupled with the services of Visma Online caused modifiability, scabalility and testability problems. These applications communicating with the services of Visma Online as service consumers showed to give several advantages such as the scalability of a single service, the reduced risk in making changes or replacing either of the components and the flexibility to move components to different hosts.

Keeping the older web applications as legacy applications tightly coupled to Visma Online instead of updating the architecture is not the best strategy for future use due to the fact that they may very well change in the future, several times. Visma has changed the layout of their webpages several times already constantly keeping it up to date and with a modern feel to the user interface. Visma Online is also constantly being extended with new functionality.

This is really not a yes or no question, what one needs to ask is if the upfront cost of the changes is worth it to make future modifiability less costly and risky?

6.1.1 Where is the system currently tightly coupled?

This question could rather have been formulated as where the system is tightly coupled in a problematic manner. The system is tightly coupled everywhere. In some legacy applications the entire application is one layer or silo, others are layered applications but within the layers tight coupling occurs.

The problematic tight coupling that occurs between Visma Online, which should be a service provider in Vismas SOA architecture and the applciations Sms, CreditCheck and Backup. The tight coupling is right now between the applications silos of Visma Sms, Backup, Creditcheck and mainly the packages within Visma Online.

6.1.2 Is it possible to identify anti-patterns with regard to coupling

There are several anti-patterns when it comes to coupling, but more interesting for this project is anti-patterns when it comes to SOA and coupling. Patterns discovered during my work that are general is the point-to-point pattern. Patterns I discovered regarding SOA is the Application Silo pattern and using SOA as EAI 2.0 [16]. I discovered the Dependency Inversion Principle being broken a couple of times. I have shown one example in figure 4.11.

There are some anti-patterns used which leads to this tight coupling. What we can see when looking at the overall software architecture of Visma online are some basic design principles broken as well as some anti-patterns when it comes to SOA being used. What we can see when diving deep into the code of the application silos is object oriented design principles being broken.

6.1.3 What are the problems caused by this tight coupling

Most of the problems can be connected to modifiability. It is not possible to scale the physical space of only one of the applications or Visma Online seperately since they are bounded together by coupling. There are problems with testability. Testing new features of Visma Online means shutting down the tightly coupled applications for testing as well.

This coupling causes problems right now such as the code being less understandable due to complexity, testing slowing down, physical scaling of a single application or service not being possible. There are also some future problems that can be caused by tight coupling.

6.1.4 What design patterns can be used to solve the tight coupling?

Some coandidates are: N-tiered pattern, layered pattern, Service composition, API gateway/broker pattern, hub and spoke pattern and the enterprise service bus pattern. The final suggestion was a combination of using SOA, small decoupled services and the API gateway as communication between service providers and service consumers.

6.1.5 What are the benefits and trade-offs of a more loosely coupled architecture?

Loose coupling should have the same advantages and disadvantages in between services as in any other part of an architecture so this should hold just the same in a SOA as in an OO architecture.

The benefits are that changes can be made more easily, changes to service contract, changes to the views, changes internally in the services can all have less impact on connected components the less coupled they are. The physical scaling can also be more

precisely pointed towards the part of the system that needs more space. If needed some components will be able to be moved to third party hosts or hosts on different locations. The components will be reusable enterprise assets.

Trade-offs are that the architecture becomes more complex and less understandable, which leads to higher demands on the employees in order to not break the principles. Adding layers like the API gateway may also serve as a bottleneck for performance and a single point of entry may be a security risk if not designed carefully, so designing the architecture takes time. Remodelling a system to conform to a new architecture takes time as well and is rather costly.

6.2 Final words about the future of Visma Online

At the rate Visma Spcs has been growing it is not that far in the future that Visma Online might be a gigantic code base which is hard for the team, especially new members, to keep track of. Keeping code that is not used is only adding to the complexity of Visma Online. I have no doubt that decoupling the applications I have been looking at is the best long-term strategy. There are several advantages in the long run to have less coupling within the project. What might cost more than it is worth is making a total remodelling of the architecture at once. It does take time to get all the loose coupling done and during that time there is really no new Use-Cases being implemented, only making sure the old ones still work. An iterative approach like Focus or the approach already being used by the Online team have advantages as well [9]. By making all newly added or newly changed code conform to the new architecture there is time to be saved and time is money. By conforming to the architecture while already doing changes and realising Use-Cases the results can be seen by all stake holders at the same time as preparing for the future complexities of the system, hopefully saving time in the future as well.

Even if the usage of SaaS was a bit out of scope for this project and I do not have access to the code of the SaaS applications, I want to point out that the usage of SOA and the usage of microservices says nothing about the delivery model. The delivery model could very well be cloud based when using SOA with an API gateway for API management. The advantages and disadvantages of this is however something for future research.

To sum this up smaller less coupled services could be achieved by using an API gateway together with a microservices-inspired strategy much of the complexity involved in the loosely coupled services could be avoided. This type of architecture would be a reasonable end goal for a large B2B company like Visma Spcs. However making a large budget to suddenly change all existing code to follow new standards might not be the most cost efficient approach, an iterative approach changing things while we are flying might be a more reasonable approach for all stakeholders involved. My case study at Visma Spcs has shown that this is actually what they have been doing and are doing, for good reasons. Since their architecture is evolving they have so far avoided situations that might turn the system into a sinking ship. All things considered this is a smart approach in this constantly changing world, as long as there is a goal in sight.

6.3 Further research

More case studies in the subject of loose coupling between services or specifically in SOA could show if the usage of services that Visma Spcs has adopted is a common strategy or pattern. I could imagine that there are more interesting strategies of how to adopt services

in other organizations that are worth doing research about. Even if the silver bullet for software architecture might never be found there is always need for finding more cost efficient patterns and strategies.

References

[1] T. E. , *et al.*, *Next Generation SOA - A Concise Introduction to Service Technology and Service-Orientation*. Upper Saddle River, NJ: Prentice Hall, 2015.

[2] S. Nimmons. Anti pattern: Tightly coupled integration. Business, Technology and Innovation Insights and Analysis – by Steve Nimmons. [Online]. Available: http://stevenimmons.org/2012/01/anti-pattern-tightly-coupled-integration/

[3] L. Newman, *Building Microservices*. Gravenstein Highway North, Sebastopol, CA: O'Reilly Media, Inc, 2015.

[4] L. B. , *et al.*, *Software Architecture in Practice*, 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

[5] Visma spcs forr och nu. https://vismaspcs.se/kontakta-oss/om-visma/visma-spcsx-forr-och-nu. Visma Spcs.

[6] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecure," *IEEE Software*, 2005.

[7] P. Bengtsson *et al.*, "Architecture-level modifiability analysis (alma)," *The Journal of Systems and Software 69 (2004) 129–147*, 2002.

[8] D.Krafzig, K.Banke, and D.Slama, *"3.5 Tight Versus Loose Coupling," in Enterprise SOA: Service-Oriented Architecture Best Practices*, 1st ed. Prentice Hall, 2004.

[9] L. Ding and N. Medvidovic, "Focus: A light-weight, incremental approach to software recovery and evolution," 2001.

[10] K. Kazman, "Atam: Method for architecture evaluation," 2000.

[11] R. C. Martin, "Design principles and design patterns," 2000.

[12] W. J. Brown *et al.*, *AntiPatterns - Refactoring Software, Architectures, and Projects in Crisi*. John Wiley Sons, Inc, 1998.

[13] F. P. Brooks, "No silver bullet - essence and accident in software engineering," 1986.

[14] "Iso/iec/ieee 24765:2010 systems and software engineering — vocabulary," 2010.

[15] M. Linssen, "Integration architecture," 2009.

[16] Heller, "Soa anti-patterns: How not to do service-oriented architecture," *Oracle White*, 2010.

[17] T. Eerl *et al.*, *SOA with REST*, 1st ed. Prentice Hall, 2013.

[18] Z. Král, "Crucial service-oriented antipatterns," *International Journal On Advances in Software*, vol. 2, 2009.

[19] L. G. Williams and C. U. Smith, "Pasa: A method for the performance assessment of software architectures," 2002.

[20] H. Runeson, "Guidelines for conducting and reporting case study research in software engineering," *Springerlink.com*, 2009.

[21] R. Kazman, G. Abowd, and P. C. Len Bass, “Scenario-based analysis of software architecture,” *IEEE Software*, 1996.

[22] D. Woods *et al.*, “Enterprise-class api patterns for cloud mobile,” 2012.

[23] D. Baum, “A comprehensive solution for api management,” 2015, an Oracle White Paper.