

Dealing with changes to shared software components

by

Roland van Dijk

Thesis submitted in partial fulfillment for the degree of
Master of Science in Business Information Technology

UNIVERSITY OF TWENTE.



Supervisors University of Twente

dr. Klaas Sikkel

dr. Chintan Amrit

External supervisors

ir. Jasper Laagland

ir. Johan te Winkel

Preface

This thesis is the end product of my research project conducted at Topicus FinCare, a software company located in Deventer in the Netherlands. Starting in August 2012 I have been working on this project, with which I will conclude my master Business Information Technology at the University of Twente. The thesis is, in a nutshell, about dealing with requirements change in an agile, code sharing environment. But more on this in the next 100 or so pages...

I'm very thankful for the opportunity and freedom I got in the way I wanted to execute this project and I hope the results in this thesis are of use for FinCare and Topicus, now or in the near future. Thanks to my supervisors at Topicus FinCare, Jasper and Johan, for their open mindset, constructive thinking and the spirited, friendly discussions. Also many thanks to Klaas and Chintan, my supervisors at the university, for always giving honest and constructive feedback and their encouragements. I wish good luck to all of you and enjoy reading!

Roland van Dijk
February 2013

Management summary

Problem Software companies more and more reuse (parts of) software for a faster time-to-market. Reusable components thus are valuable assets for any software company. However, designing for reuse requires upfront investment and may not directly show financial benefits. Moreover, when a company employs an agile development culture, deadlines are tight and come in rapid succession. To continuously assure a high level of quality, it is desired to have a stable product each time a new version of the product is released. Stable products require stable components, which can be a paradox when employing an agile development approach where software is built in short and frequent iterations.

Purpose The goal of this study is to investigate how Topicus FinCare, the focal subject of this study, can mitigate risks associated with changes to shared components.

Results From a literature study we abstracted a total of 28 risks of which we have found 7 of relevancy for FinCare, categorized in 3 different solution domains. In the domain of *dealing with technical implications* (1) maintaining stability of functionality for core platform components and (2) the chance of some change unintentionally rippling to another codebase/application are the main issues. In the domain of *dealing with organizational culture* (3) usage of an immature platform in production, (4) a lack of reusability of components and (5) technical debt are the main issues. Finally, in the domain of *dealing with communication and collaboration* (6) having a bottleneck in the development process and (7) a lack of communication about updates are the main issues.

Recommendations Based on interviews at Topicus and additional literature we provide a large number of best practices and mitigation approaches in our study, of which the following are of direct value for FinCare and can be implemented most easily:

- Involve more people in the development of the shared codebase
- Have a simple, but clear communication policy when modifying components
- Occasionally let developers and analysts switch projects

Besides the above ‘low hanging fruits’, we found three important topics which will become important for FinCare in the near future and which we found already relevant for the interviewed units at Topicus. We recommend both Topicus and FinCare to:

- Develop a simple but consistent component development model where billing work effort on shared components internally (across teams) is fair and externally (to customers) is transparent
- Invest in visualization support for component usage by projects and applications
- Within business units employ a social self-regulating codebase, without strict code ownership, where everyone can contribute and is responsible for their own modifications. Between business units, follow a producer-consumer paradigm.

Contents

Preface	iii
Management summary	v
List of figures	xi
List of tables	xiii
1 Introduction	1
1.1 Project background	1
1.2 Definitions	2
1.2.1 What is a product platform?	2
1.2.2 What is a software component?	3
1.2.3 What is a shared codebase?	3
1.2.4 What is a SaaS delivery model?	3
1.2.5 Summary	4
1.3 Project scope	5
1.3.1 The project's case: health care claims platform	5
1.3.2 Codebase dependencies	5
1.3.3 Product-line strategy	6
1.3.4 Growth strategy of Topicus	7
1.3.5 Summary	7
1.4 Problem statement	7
1.5 Relevancy for Topicus	8
1.5.1 Unit A	8
1.5.2 Unit B	8
1.5.3 Unit D	8
1.5.4 Summary of issues	10
1.6 Project objectives	10
1.7 Research questions	10
2 Research approach	13
2.1 Introduction	13
2.2 What is the risk of a changing requirement?	13
2.2.1 Risks in general	13
2.2.2 Software project risks	14
2.2.3 Requirements risks	15
2.2.4 Defining requirement changes	15

2.2.5	Requirements change risks and project performance	17
2.2.6	Conclusion	19
2.3	What is working on a shared codebase?	20
2.3.1	Development structures	20
2.3.2	Shared codebases challenges	22
2.3.3	Challenges of working with a shared codebase	22
2.3.4	Impact analysis	25
2.4	Research approach	27
2.4.1	Research questions	27
2.4.2	Research construct	28
2.5	Conclusion	29
3	Literature study of case studies	31
3.1	Introduction	31
3.2	Research approach	31
3.2.1	Define scope	31
3.2.2	Identify fields of research	32
3.2.3	Define search terms	32
3.2.4	Search	32
3.2.5	Filter out doubles	32
3.2.6	Cut down sample based on title+abstract	33
3.2.7	Cut down sample based on full text	33
3.3	Results	33
3.3.1	Volatile market	34
3.3.2	Client influence	34
3.3.3	Market pressure	34
3.3.4	Business strategy	35
3.3.5	Ambiguity	36
3.3.6	Scope	37
3.3.7	Scattered functionality	38
3.3.8	Development of reusable components	38
3.3.9	Communication and knowledge sharing	40
3.3.10	Experience in reuse	41
3.3.11	Adoption a PLE approach	41
3.4	Derived risks	42
3.4.1	Risk of operating in a volatile market	44
3.4.2	Risk of stakeholder influence	44
3.4.3	Risk of time-to-market pressure	44
3.4.4	Risk of evolving standards	44
3.4.5	Risk of political aspects	45
3.4.6	Risk of business philosophy focusing on short-term goals	45
3.4.7	Risk of business value thinking	45
3.4.8	Risk of prioritizing of mainstream product	46
3.4.9	Risk of changing the business strategy	46
3.4.10	Risk of reusing immature components	46
3.4.11	Risk of unclear requirements	46
3.4.12	Risk of different interpretations of artifacts	47
3.4.13	Risk of goal ambiguity	47
3.4.14	Risk of scope widening	47

3.4.15	Risk of scattered functionality	47
3.4.16	Risk of delocalized plans/documents	48
3.4.17	Risk of iteratively changing reuse components	48
3.4.18	Risk of changes in product line assets at the product level	48
3.4.19	Risk of enhancement to a cross-cutting concern	48
3.4.20	Risk of component granularity	48
3.4.21	Risk of circular dependencies	49
3.4.22	Risk of non-standardized configuration interfaces	49
3.4.23	Risk of early binding of build-level dependencies	49
3.4.24	Risk of making a composition by hand	49
3.4.25	Risk of making an application/component reusable	50
3.4.26	Risk of heterogeneous communication	50
3.4.27	Risk of centralization in group based collaboration networks	50
3.4.28	Risk of reuse experience level	50
3.5	Conclusion	50
4	Interviews	51
4.1	Introduction	51
4.2	Goals	51
4.3	Designing the interview	51
4.3.1	Why interviews?	51
4.3.2	Interview type	52
4.3.3	Interpreting data	52
4.4	Interview protocol	53
4.4.1	Shared codebases challenges and requirements change	53
4.4.2	Heuristic	53
4.4.3	Interview questions	54
4.5	BTOPP-model	54
4.6	The interviewed business units	56
4.6.1	Unit A	56
4.6.2	Unit B	56
4.6.3	Unit C	56
4.7	Interview results	57
4.7.1	Unit A	57
4.7.2	Summary unit A	62
4.7.3	Unit B: issues	64
4.7.4	Unit B: solutions	69
4.7.5	Summary unit B	74
4.7.6	Unit C	76
4.7.7	Summary unit C	79
4.8	Conclusion	80
5	Casestudy: risks at FinCare	81
5.1	Introduction	81
5.2	Bottom-up codebase analysis	81
5.2.1	FinCare Products	81
5.2.2	Teams and responsibilities	82
5.2.3	Development process	82
5.2.4	Architecture of FinCareClaim	83
5.2.5	Technical architecture	83

5.2.6	Modules, components, packages and libraries	84
5.2.7	Visualizing the repositories	85
5.2.8	Conclusions	91
5.3	Relevant risks	97
5.3.1	Introduction	97
5.3.2	Discussion of risks	97
5.4	Conclusion	101
6	Finding suitable mitigation approaches	103
6.1	Introduction	103
6.2	Mapping solutions/best practices to shortlist	103
6.3	Discussion of mitigation approaches	106
6.3.1	Questionnaire	106
6.3.2	Dealing with technical implications	107
6.3.3	Dealing with organizational culture	110
6.3.4	Dealing with communication and collaboration	113
6.4	Conclusion	118
7	Conclusion	119
7.1	Answering the problem statement	119
7.2	Things you already can implement tomorrow	121
7.3	Things to implement in the long term	121
7.4	Scientific relevancy	123
7.5	Validity	124
	Appendices	125
A	Challenges versus change characteristics	127
B	Interview protocol	129
C	Questionnaire	131
	Bibliography	139

List of Figures

1.1	Claim process	5
1.2	Codebase dependencies	6
1.3	Frameworks at unit D.	9
2.1	Project-driven development.	20
2.2	Component-driven development.	21
2.3	Product-driven development.	22
2.4	Feature-driven development.	23
2.5	Types of impact analysis methods abstracted (Lehnert, 2011).	26
2.6	Research construct.	29
3.1	Volatile market.	34
3.2	Client influence.	35
3.3	Market pressure.	35
3.4	Long-term versus short-term PLE.	36
3.5	Asset ambiguity.	37
3.6	Scope widening and requirement complexity increase.	37
3.7	Scattered and delocalized functionality.	38
3.8	Change propagation.	39
3.9	Knowledge questions.	39
3.10	Dispatcher role.	40
3.11	Knowledge sharing and reuse champion.	41
4.1	BTOPP model	55
4.2	Codebase unit A	57
5.1	High-level architecture of the FinCareClaim platform	84
5.2	Complete dependency graph of codebase at FinCare	89
5.3	Dependencies between assemblies	90
5.4	NuGet dependencies between repositories	91
5.5	Complete codebase dependencies of the FinCareClaim platform	92
5.6	FinCareClaim platform grouped per application	93
5.7	Logical coupling of FinCare repository	94
5.8	Logical coupling of FinCare repository grouped by FinCareClaim applications	95
5.9	Logical coupling older than 1 year ago	96
6.1	Causal relation diagram of dealing with technical implications	107
6.2	Causal relation diagram of dealing with organizational culture	111

6.3 Causal relation diagram of dealing with communication and collaboration issues 114

A.1 Reuse challenges vs change characteristics 128

List of Tables

2.1	Risk categorization	14
2.2	Software change factors.	16
2.3	Software change factors according to theme.	16
2.4	Characteristics of requirement changes according to McGee (McGee, 2011).	18
2.5	Characteristics of requirements changes according to Williams et al. (Williams and Carver, 2010)	19
3.1	Derived risks from industrial case studies	43
4.1	Relation topics and interview directions	54
4.2	Interview results: issues	63
4.3	Interview results: solutions or best practices	64
4.4	Interview results: desires	64
4.5	Interview results: issues unit B	75
4.6	Interview results: solutions or best practices unit B	75
4.7	Interview results: desires unit B	76
4.8	Interview results: issues unit C	79
4.9	Interview results: solutions unit C	80
4.10	Interview results: desires unit C	80
5.1	Mercurial repositories at FinCare	85
5.2	Collapsed FinCareClaim-platform repository	85
5.3	Color mapping	86
5.4	Shared packages with degree higher than 1	88
5.5	Shared assemblies of FinCareClaim	88
5.6	Risks according to Likelihood (L), Impact (I) and Relevancy (R)	102
6.1	NPS of problems dealing with technical implications	108
6.2	NPS of solutions dealing with technical implications	109
6.3	NPS of problems dealing with organizational culture	111
6.4	NPS of solutions dealing with organizational culture	113
6.5	NPS of problems dealing with communication and collaboration	115
6.6	NPS of solutions dealing with communication and collaboration	117

- Chapter 1 -

Introduction

This research project has been carried out for Topicus FinCare, a software development company located in the Netherlands. In this chapter we will introduce the company, give some basic definitions, illustrate the research problem and formulate the problem statement.

§ 1.1 PROJECT BACKGROUND

Topicus FinCare is an autonomous business unit from the PBT Holding (from now on just Topicus) and has approximately 360 employees and 16 of such autonomous business units which are run as investment centers. Topicus operates in all kinds of domains, including government, health care, education and finance.

FinCare has 22 employees and consists of small software development teams, each working on their own projects. They follow a lean, or ‘agile’ software development approach, which means they build software in multiple iterations of approximately 4 weeks with few documentation and close customer involvement. Each iteration results in a working prototype or sub-system of the product which is demonstrated to the customer in order to check if the product is in compliance with the customer’s wishes.

This project stems from a desire of FinCare to continuously assure code quality in a growing organization which values their lean, agile culture very highly and has a natural tendency of rejecting organizational overhead in order to achieve code quality assurance. This seemingly paradoxical situation is the reason FinCare issued this project. They know they will evolve over time, they know managerial ‘overhead’ might be inevitable if the organization grows bigger, but is there a path where software quality is assured without making concessions to the existing development culture? The situation is best illustrated by three examples which are typical for FinCare projects:

First example Let’s assume we have multiple teams work on different projects, but they all use functionality from a shared codebase. The customer of one of these teams has the desire for some new functionality. In order to implement this the development team has to modify core components located in their product platform. How do they make sure that this new functionality doesn’t interfere with the product-specific components of the other solutions offered by their SaaS delivery model?

Second example Assume a cost estimation has to be made for some new or changing feature to a platform module. How can FinCare make sure they know all the work that has to be done

in advance? For FinCare knowing this is important since they mostly work with a fixed price arrangement. Currently FinCare sometimes performs impact analysis to estimate the costs and the work effort, but this gives no guarantees and still a lot of unforeseen work arises during the project's execution.

Third example At any moment in the development process new or changing requirements may surface which need to be included in the development cycle. In the case of new requirements at FinCare sometimes the cycle starts at the information analyst who identifies together with the customer the requirement, translates it into a functional specification which is then further analyzed and translated into a technical specification. Sometimes however, there is no technical specification made and the programmer starts coding right away. In the case of changing requirements, the first two steps sometimes are skipped completely and a programmer directly starts working on the implementation, without any functional specification. This of course can lead to unforeseen problems. Perhaps it introduces undesired behavior in other components, or it may conflict with existing (implemented or not) requirements.

In the examples a number of terms are used like *platform*, *component* and *shared codebase*. Before we continue with describing the research problem, we give definitions of these terms.

§ 1.2 DEFINITIONS

1.2.1 What is a product platform?

Designing software in the very traditional way starts with defining requirements, creating a high-level design and creating an architectural design before starting to work on the implementation. The implementation process then encompasses creating the elements which are defined in the architecture of the software solution.

A **software architecture** then is defined as: “*the structure or structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them*” (Bass et al., 2003).

Nowadays, software is more and more created from existing software parts. Reusing previous created software assets to create new software is intuitively more efficient than starting from scratch. When a family of products is created based on a common base of components, we speak of a **product family**. The common base of components then is called a **product platform**.

A product platform is defined as: “*a set of subsystems and interfaces intentionally planned and developed to form a common structure from which a stream of derivative products can be efficiently developed and produced*” (Muffatto and Roveda, 2002).

According to Muffatto and Roveda a product platform is strongly related to product architecture: “*product architecture influences the development of a platform since, the more integrated the product architecture is the more model-specific its interfaces are and it is more difficult for its subsystems to be shared with other models of a family [...] the product platform is a means to give architectural complexity to products while contemporary exploiting some advantages of modularization and standardization*” (Muffatto and Roveda, 2002).

For example, a bicycle manufacturer can have a cheap, a medium and high-end bicycle design based on the same basic frame. The architecture of the bicycle describes what component goes where and how the components relate to each other. Where do the lights go, where does the saddle go, etc. The product platform then are all the components and how they relate to the frame and the functioning of the bicycle. The product family are all the different types of bicycles manufactured using the product platform.

In software development employing a product platform is called **software product line engineering** (SPLE) (Pohl et al., 2005). The advantages of SPLE with respect to one-off development are shortened time-to-market, increased product quality and decreased costs. Most SPLE approaches have a two-phased approach where first domain artifacts are created (domain engineering) after which application development begins to roll out different applications supported by the domain artifacts.

1.2.2 What is a software component?

A software platform consists of assets which can be reused in other applications. Such assets is in general what we call **software components**. More formally a software component is defined as: “*a distinguishable, inter-changeable piece of software which offers a coherent set of functionality*”. Or: “*A software component is a unit of composition with explicitly specified provided, required configuration and quality attributes*” (Bosch, 2000). Another definition is given by Szyperski et al.: “*software components are binary units of independent production, acquisition and deployment that interact to form a functioning system*” (Szyperski et al., 2002). For this study we will use the first definition of a component as stated by Bosch.

A software component is something different than a module. A **module** is defined as follows: “*Modules are units of implementation. Modules represent a code-based way of considering the system. They are assigned areas of functional responsibility. There is less emphasis on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as: What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use?*” (Bass et al., 2003).

In this thesis we will often use the term **reusable software components**, which basically are components (as defined above) which can be deployed as part of a product platform. Note that there is a difference between domain artifacts from SPLE and reusable software components in general. The first are components designed with the purpose to be deployed in a specific product family as part of a product line engineering process. The latter is designed to be reusable in general or in the context of a domain, not necessarily with the purpose to be part of a specific product family.

1.2.3 What is a shared codebase?

A **shared codebase** is nothing more than those software components which are shared among applications of a software development company. In practice a shared codebase is a mixture of external developed components and internal developed components. To allow multiple teams to simultaneously work on the shared codebase, collaborative software development nowadays requires the use of repositories with version control (like GIT, SVN, CVS or Mercurial). This enables developers branching of code, merging of deltas and reverting changes.

1.2.4 What is a SaaS delivery model?

After software is written, it has to get delivered to customers. There are a number of variations here as how to achieve this, we will only shortly describe the main categories.

On-site software Software you buy on a disc and install on your PC falls under this category. but also large ERP systems installed on some system. On-site software is software which is installed and operated at the location of the customer.

Off-site hosting Software accessible (via the Internet) from a centralized location which is maintained by some third party is off-site hosting. Application Service Providers (ASPs) are a variant of this delivery model.

Software as a service (SaaS) Software as a Service is a delivery model *‘focused on exploiting economics of scale by offering the same instance of an application (or parts thereof) to as many customers, called tenants, as possible.’* (Mietzner et al., 2009). There are 3 variants of SaaS. First, with **single instance** *‘all tenants use the same instance of the SaaS application. With same instance is meant the same work flow, using the same code on the same infrastructure’* (Mietzner et al., 2009). Second with **single configurable instance** *‘all tenants use the same instance of the SaaS application. However, the instance is configurable on a per-tenant basis. Run-time configurations through configuration meta-data, e.g. configuration files’* (Mietzner et al., 2009). Third, with **Multiple instances** *‘each tenant uses a different instance of a service This requires separate code for each tenant, while it allows for more flexibility for customer requirements’* (Mietzner et al., 2009).

1.2.5 Summary

Below is a summary of the definitions discussed in the previous sections.

Product platform A product platform is the abstract term for the collection of generic software assets, their purpose and interrelations, which can be extended and reused to compose new software applications.

Shared codebase A shared codebase is the actual collection of assets shared among different software applications. A codebase repository is then is a digital working environment where software developers can commit modifications to the shared assets and retrieve updates from.

Component A component, or software component, is a distinguishable, inter-changeable piece of software which offers a coherent set of functionality.

Software as a service Software as a Service or SaaS is a delivery model of software where customers are tenants of the same software solution. The solution can be configured for individual wishes of tenants, but the core functionality in essence is the same for everyone. The components which form and enable this functionality is what is called a SaaS platform.

Module A module is a unit of implementation. For example, in a SaaS platform a tenant can have specific wishes regarding functionality. Functionality is then grouped in modules, for example ‘user management’ for managing access rights of users or ‘financial reports’ for an accountancy product. Modules can then switched on or off to give customers some level of choice. Also, modules often have some level of configuration to customize the behavior of the module.

Product-specific component A software platform can support multiple solutions, which all can be SaaS platforms on their own. A product-specific component then is a component which only contains functionality in the scope of a single or a subset of the solutions supported by the platform.

§ 1.3 PROJECT SCOPE

The examples illustrated in section 1.1 indicate a wide variety of issues related to changes during software development in a setting where code is shared among projects and/or teams. However, the given examples are very broad and generic, so we introduce a recent actual project at FinCare which will be used as a case throughout this study. Throughout the study we will use fictitious names for all products and customers.

1.3.1 The project's case: health care claims platform

FinCare focuses on claims of health care products. In the Netherlands every citizen is obliged to have a health insurance at a health insurance company. Dutch law enforces a free insurance market meaning that clients must be able to switch between health insurance companies when their contract expires. Health insurance firms therefore offer contracts of mostly 1 year after which the client can switch if he/she desires so. Clients can choose between multiple levels of insurance determining what treatments or products are covered by the health care plan. If a client consumes a health product the health care provider usually doesn't get the money directly from the client. If the client's health care plan covers the consumed product the health care provider can claim the treatment at the insurance company. This is the market where FinCare operates. They provide a platform which receives claims of care providers, processes them and passes them to the health insurance company. The insurance company then can reject or requests additional processing by the platform. This process is depicted in Figure 1.1.

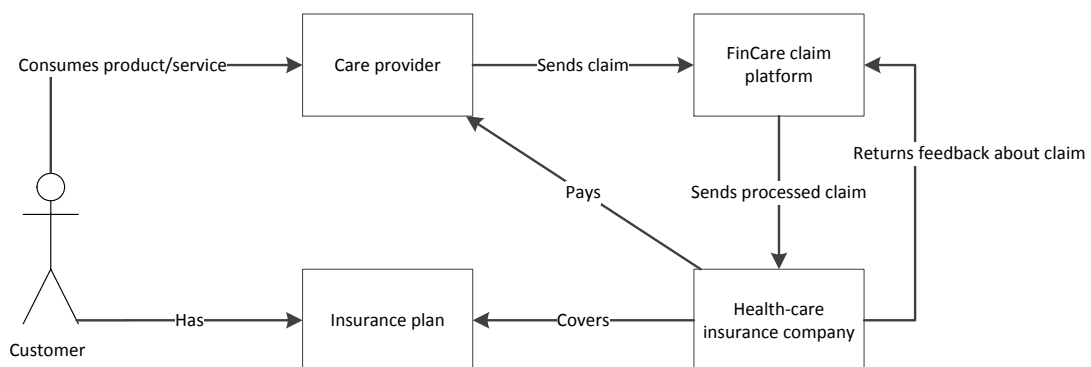


Figure 1.1: *Claim process*

Currently FinCare has four separate products live in production for their corresponding customers Careco and Medico. Both are intermediary firms which have a client-base of health care providers. Careco focuses on so called ‘care groups’ and Medico focuses on pharmacies. A care group can best be described as an alliance between different health care providers around a specific treatment. The partners in these groups have arrangements for specific treatments and receive a fixed amount of the total sum reserved for this treatment.

1.3.2 Codebase dependencies

The applications for Careco and Medico (called CarecoSoft and MedicoSoft respectively) both run on different servers, but these servers contain instances which consist of modules from a common codebase as well as application specific code. This is because both applications have common functionality like for example sending invoices or health care provider management.

Their common functionality is bundled together in a platform called FinCareClaim. The common codebase consists of two layers. The first layer is the claim processing platform (FinCareClaim), the second layer a collection of components used throughout FinCare. The second layer contains two types of components, the so called ‘Force’ components and generic external components. Both types are shared among all projects within FinCare. The Force components some years ago were shared among multiple business units within the Topicus Holding, but for FinCare this is no longer the case. They made their own copy of them and now have their own version, altered such that merging them back is impossible. The dependencies are illustrated in a simplified manner in Figure 1.2. Later on in this study the codebase is analyzed in more detail.

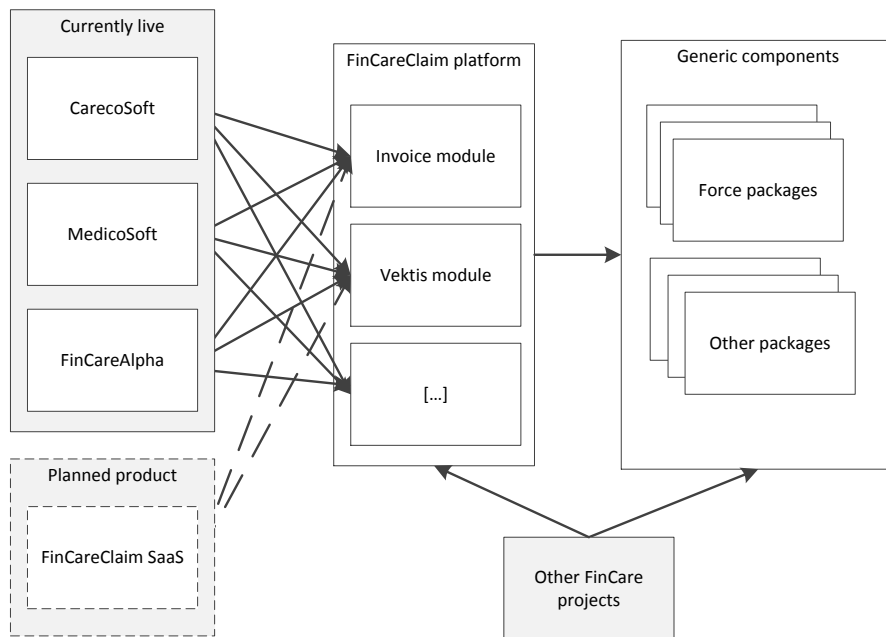


Figure 1.2: *Codebase dependencies*

1.3.3 Product-line strategy

FinCare is planning to expand their client-base by serving their platform to independent treatment centers (outpatients clinics), first-line treatment centers (independent complexes containing GP offices, physical therapy, etc.), geriatric rehabilitation centers and regional GP offices. Since these clients are too small to pay for their own implementation, FinCare is developing a SaaS solution for these clients called FinCareClaim SaaS.

Because FinCareClaim SaaS will be a SaaS solution, there is no need to compile a separate instance of the claim platform and its modules. However, they will require a different front-end, different data structures and perhaps some specific modules. FinCare expects that their current framework will support this, but FinCare already experienced in other products that specific product functionality can conflict with or introduce undesired effects in other applications. The chance of this happening only increases when more teams and applications start using the shared codebase.

1.3.4 Growth strategy of Topicus

Topicus has the philosophy that when a company grows too large, they lose their innovative edge and productivity because of organizational overhead required to manage the development process. Therefore they apply a general rule that whenever a business unit grows beyond 25 employees, they should split up in separate business units. FinCare has grown in three years from a few employees to around 20 employees. Chances are high that in the next three years they will split up. A potential problem with this is code ownership of code entities in a shared codebase. Developers one day working physically together on shared functionality can become physically separated and start working on totally different projects. When new functionality is introduced which requires refactoring some parts of a shared codebase, how do developers and business analysts know where to find the knowledge to conduct an impact analysis? How are the responsibilities managed of these parts of the code? Instead of working together on the general Force modules, FinCare made a copy of these components when they started as a business unit and developed their own version which was never merged back with the original repository. One could argue that this was a valid decision since the business units work on very different products, but it may very well be the case that both business units develop code which for both are valuable or even work on the same functionality without knowing from each other.

1.3.5 Summary

From the examples and background information above we can derive the following potential issues for Topicus FinCare:

- In a software development environment with multiple teams, possibility spanning multiple business units, working on products which make use of a shared codebase, a change origination from a requirement change in one project can have a ripple effect to other projects
- When not knowing the impact of a requirement change, it can introduce undesired defects
- Unforeseen work effort because of unforeseen effects can result in a too low cost estimation, resulting in less financial margins for a release of a product.
- When splitting up FinCare in new business units in the future, not managing code ownership and responsibilities of the shared codebase can result in repositories deviating from each other with both business units managing their own version of the once shared codebase

§ 1.4 PROBLEM STATEMENT

Based on the issues outlined above, the following problem statement is defined for this project:

Problem statement: How can FinCare mitigate risks associated with requirement changes in an agile development environment where multiple projects share functionality located in a shared codebase?

§ 1.5 RELEVANCY FOR TOPICUS

The problem statement is derived from the situation at FinCare, but these problems are also relevant for other business units of Topicus. Three business units within Topicus were contacted and asked how shared codebases play a role in their daily work to find out if involving them would be relevant for this study. Below a small description of the different units and the situations they face.

1.5.1 Unit A

Unit A develops and maintains 4 large educational tracking systems for primary education, secondary education and higher education. These systems are used for administration, rostering, grading and other back-office activities. Two systems have a shared codebase, the other applications all have specific business logic. All applications do however make use of the so called ‘COBRA’ codebase, which contains very generic components, but no business logic. Also, the dependencies of all applications are managed by a shared component, which functions as a configuration manager. Unit A employs the culture that everyone (all units of Topicus) can use COBRA components, and everyone at unit A can commit modifications on COBRA and modify components of COBRA. Also, unit A has the policy that at the end of the day a snapshot of the current code is deployed in production. Of course they have a build server which tests for errors using regression tests, but from time to time bugs propagate to other systems and are not always noticed by developers. Besides this error prone situation, they have to maintain a lot of interfaces with external systems (mostly government systems). From time to time API’s change on the side of these systems without proper up-front communication, resulting in unstable products.

1.5.2 Unit B

Unit B develops health care chain applications for a divers set of health care providers. They have a centralized component base with for example functionality to generate forms out of a data model. Mainly the centralized component base contains front-end logic, but there also some shared business logic components. Currently they have three systems using this framework, all with a different customer base and all with planned expansions of their customer base. Two other business units also make use of these components, moreover, they have commit privileges on the repositories. From time to time they cooperate in refactoring components or in adding new functionality. Mostly the party who is in need of the refactoring or new functionality takes the initiative and invests in the effort. Unit B has had multiple occasions where modifications on their shared codebase impacted other products and it always a difficult to assess what will change in advance or to include all contingencies from other projects or business units.

1.5.3 Unit D

Unit D develops mid-office and back-offices for mortgage lenders or intermediaries. Unit D has customers exclusively for a mid-office or back-office application, but also for both. In their products they make a distinction between the Core product and the Shell. The core contains the actual business logic of the application, while the shell consists of modules containing interfaces, the front-end or other domain specific functionality. Unit D started with offering a mid-office application and gradually extended their customer base. At one point they gained the prospect of serving a high value customer, while their framework (supporting new extensions) was not yet stable. Unit D decided to continue developing on the framework while also starting to work on

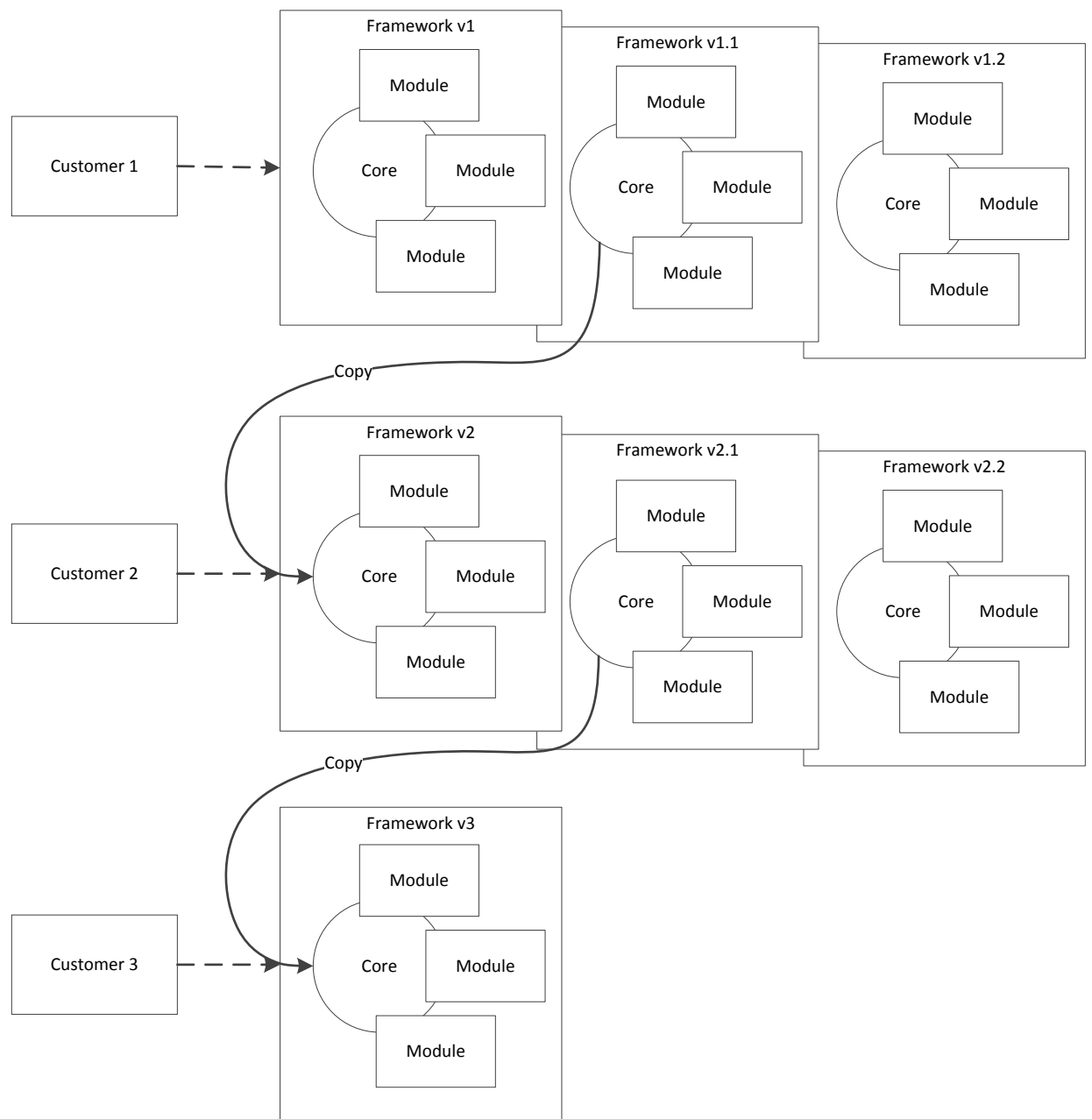


Figure 1.3: *Frameworks at unit D.*

a copy of the framework to serve this high value customer. To prevent too much architectural deviation unit D initiated a project to develop a stable framework merging all features of both frameworks. However, another high prospect client was already knocking on the door. Unit D currently is facing the challenge: do we pursue getting this client and start developing yet on another framework, or do we wait and let the framework mature first? The situation is depicted in Figure 1.3. The modules in the shell are often very client specific, their customers are rather big and demand tailored integration in their IT landscape. Putting this kind of functionality in the Core therefore is very hard.

1.5.4 Summary of issues

From the situations described above we can derive a number of issues relevant for Topicus:

- Code/component deviating from their intended purpose
- Deciding what to refactor for reuse
- Who is responsible for shared components?
- Managing dependencies among products
- Planning for additional products for which the frameworks should be used (or which the framework should support)
- Testing strategy of core components
- Developing framework for specific customers

The issues at this stage were only collected during informative talks at the units, so no formal data was yet collected. What we observed is that the FinCare, unit A and unit B share more or less the same culture, but unit D is very conformed to formal processes (because of the nature of their customer-base) and hence has a complete different culture. For the remainder of the study, we keep the case of unit D in mind but we will not be focusing on exploring their situation further.

§ 1.6 PROJECT OBJECTIVES

The primary objective of this project is to provide a solution for the problem statement for FinCare. The problem statement asks for a risk mitigation approach. The first logical objective then would be to investigate what possible risks FinCare can be exposed to. This, of course, in the context of requirements changes in shared codebase environment. After the risks have been identified, the next logical task would be to find possible mitigation approaches for them. Without making an explicit choice as to how to achieve this, the final step would be to frame mitigation approaches in the context of FinCare and the FinCareClaim SaaS project as introduced earlier. In summary the objectives are:

- Find the possible risks a company like FinCare can face from the perspective of requirements changes in a shared codebase environment.
- Find risk mitigation approaches to mitigate these risks
- Find potential issues for the FinCare and their codebase
- Evaluate what approaches would be suitable for FinCare to implement

§ 1.7 RESEARCH QUESTIONS

The research objectives are operationalized into the following research questions:

1. What are the risks of a shared codebase environment with respect to changing requirements?
2. What mitigation approaches can be used to mitigate these risks?
3. What are the potential risks for FinCare?

4. What risk mitigation approaches are suitable to implement by FinCare to mitigate the potential risks?

In the next chapter we will formulate the research approach, based on a brief literature review of risks and general challenges with regard to working on a shared codebase.

- Chapter 2 -

Research approach

§ 2.1 INTRODUCTION

The problem statement for this research is defined as: *How can FinCare mitigate risks associated with requirement changes in an agile development environment where multiple projects share functionality located in a shared codebase?* We operationalized the problem statement with 4 research questions where we respectively **(1)** search for possible risks, **(2)** search for possible solutions, **(3)** assess what risks are relevant for FinCare and then **(4)** evaluate what solutions are suitable for FinCare. In this chapter our research approach is stated, but first we will explore terms like *risks*, *changing requirements* and *working on a shared codebase*. So in this chapter we will:

- Give a definition of risks of changing requirements
- Describe general challenges of working on a shared codebase environment
- State the research approach

§ 2.2 WHAT IS THE RISK OF A CHANGING REQUIREMENT?

2.2.1 Risks in general

According to Charette (Charette, 1989) there are two types of risks: speculative risks (or dynamic risk) and static risks. Speculative risks are risks having both profit and loss associated with them whereas static risks only have losses associated with them. According to Charette software projects have due with the first type of risks. A risk then is an event which can happen with some uncertainty and the result can be either loss or profit. This loss or profit is something which very much depends on someone's point of view. One may view a situation in one context very differently than in another context. Nevertheless, the uncertainty attribute indicates that there is a chance associated with a risk. With uncertainty we mean that some information about the system may be known but here is not enough knowledge about it to provide certainty. In this context Charette (Charette, 1989) state three types of uncertainty:

- **Descriptive or structural certainty:** The absence of information relating to the identity of the variables that explicitly define the system under study
- **Measurement uncertainty:** The absence of information relating to the assignment of a value to the variables used to describe the system

- **Event outcome uncertainty:** When the predicted outcomes and therefore their probabilities cannot be identified

The attribute of uncertainty indicates that there is always a choice associated with a risk; if there would be no choice, there would be no risk. In summary for this study we consider an event, action or thing a risk if:

- There is a loss or profit associated with it
- There is uncertainty or a chance involved
- There is some choice involved

According to Charette (Charette, 1989), a risk can be classified according to three dimensions: *probability (or likelihood of occurrence)*, *frequency* and *source*. The probability and frequency dimensions can be a numeric scale, whereas source is a nominal scale. Charette gives an example classification using the probability and source dimensions as illustrated in Table 2.1. The probability dimension here is made nominal and has the following values:

Known risks according to Charette (Charette, 1989) are ‘*those that after thorough, critical, and honest analysis of the project plan would illuminate as frequently occurring and with a high probability of currently existing.*’ He gives examples, which for software plans are then typical, as lack of system requirements, overly-optimistic productivity rates and short schedules.

Predictable risks then are ‘*those that experience dictates one may encounter with a high probability.*’ In the same context of software project plans he gives examples as lack of timely client approval, reviews and personnel turnover.

Unpredictable risks are, according to Charette, ‘*those that could happen, but the likelihood or timing of these events occurring cannot be projected very far in advance.*’ He gives examples as funding availability, poor management and acquisition strategy change.

	Known	Predictable	Unpredictable
Lack of information			
Lack of control			
Lack of time			

Table 2.1: Risk categorization

2.2.2 Software project risks

A **software project risk** is defined as ‘*a set of factors or conditions that can pose a serious threat to the successful completion of a software project*’ (Wallace et al., 2004). According to Wallace et al. we can distinguish between 6 dimensions of software project risks:

- Team risk. For example: frequent conflicts between development team members, team members not familiar with the task(s) being automated
- Organizational environment risk. For example: lack of top management support for the project, corporate politics, unstable organizational environment
- Requirements risk. For example: Incorrect system requirements, undefined project success criteria, difficulty in defining the inputs and outputs of the system
- Planning and control risk. For example: Project milestones not clearly defined, lack of ‘people skills’ in project leadership
- User risk. For example: Lack of cooperation from users, conflict between users, users with negative attitudes toward the project

- Complexity risk. For example: Immature technology, large number of links to other systems required

According to Charette (Charette, 1989) if we want to classify risks of software engineering, two main categories can be identified: those risks that deal with the process of developing software and those that deal with the product itself. Within these categories the following types of risks are defined:

- Acquisition risks: technical risks, schedule risks, cost risks, customer need risks
- Decision risks: operational risks, support risks

As one can see, the risk definitions have a fairly large overlap, but differ on the crucial point that according to Wallace et al. software project risks pose a threat to the successful completion of a project, whereas the definition of Charette also covers positive outcomes (dynamic risks versus static risks). In the next sections we look at some more risk definitions.

2.2.3 Requirements risks

In the category of requirements risks of software project risks there are different listings of what risks this category encompasses. We here give two examples of such listings.

Han and Huang (Han and Huang, 2007) discuss the following risks:

- Continually changing system requirements
- System requirements not adequately identified
- Unclear system requirements
- Incorrect system requirements

In a study by Wallace et al. (Wallace et al., 2004) the following listing is given :

- Incorrect system requirements
- Users lack understanding of system capabilities and limitations
- Undefined project success criteria
- Conflicting system requirements
- Difficulty in defining the input and outputs of the system
- Unclear system requirements
- System requirements not adequately identified
- Continually changing system requirements

In both listings requirements change is indicated as one of the risks of the ‘requirements risks’ dimension of software project risks. Alternative terms of changing requirements used in literature are *requirements volatility*, *requirements creep*, *scope creep*, *requirements scrap*, *requirements instability* and *requirements churn* (Ferreira et al., 2011). These definitions are also static risks. Next we define requirements changes.

2.2.4 Defining requirement changes

Since a requirements change is the topic of study for this project, we need to define what a requirements change is. In the previous section we indicated where in the area of software risks, requirements risks is located. We will define a requirement change according to the characteristics a change has. We first will discuss how a software change in general is defined and then discuss a number of classifications of requirements changes.

2.2.4.1 Software changes

A classification of *software changes* is provided by Buckley et al. (Buckley et al., 2005). They propose a classification according to characterizing factors and influencing factors. An influencing factor is a factor which influence the change mechanism whereas a characterizing factor captures the essence of a change mechanism. An overview of the factors is given in Table 2.2.

Factor	Characterizing factor	Influencing factor
Time of change	x	x
Change type	x	x
Change history	x	
Degree of automation	x	
Activeness	x	
Change frequency		x
Anticipation		x
Artifact		x
Granularity		x
Impact		x
Change propagation		x
Availability		x
Openness		x
Safety		x
Degree of formality		x

Table 2.2: *Software change factors.*

When organizing the factors of Buckley et al. according to theme, we get the ordering as shown in Table 2.3. The classifications complement each other and give a good indication of what information about a software change can be captured.

Theme	Factor
Temporal properties	Time of change
	Change history
	Change frequency
	Anticipation
Object of change	Artifact
	Granularity
	Impact
	Change propagation
System properties	Availability
	Activeness
	Openness
	Safety
Change support	Degree of automation
	Degree of formality
	Change type

Table 2.3: *Software change factors according to theme.*

2.2.4.2 Requirement changes

In (Nurmuliani et al., 2004) the authors classify a requirement change according to change type, the reason of the change and the origin of the change. Possible requirement changes then are *addition*, *deletion* and *modification*. Possible reasons for a requirement change according to (Nurmuliani et al., 2004):

- Defect fixing
- Missing requirements
- Functionality enhancement
- Product strategy
- Design improvement
- Scope reduction
- Redundant functionality
- Obsolete functionality
- Erroneous requirements
- Resolving conflicts
- Clarifying requirements

A classification of the source of a requirement change is given by McGee et al (McGee and Greer, 2009). The authors identify 5 change domains:

- **Market** Differing needs of many customers, government regulations
- **Customer organization** Changing strategic direction, political climate
- **Project vision** Change to problem to be solved, product direction and priorities, stakeholder involvement, process change
- **Requirements specification** Change of specification, resolution of ambiguity, inconsistency, increased understanding
- **Solution** New technical requirements, design improvement, solution elegance

In a follow-up study by McGee (McGee, 2011) a classification for requirements changes is discussed which is given in Table 2.4. Another classification of requirement changes is discussed by Williams et al. (Williams and Carver, 2010) and given in Table 2.5. All these definitions indicate that a requirement change can be really anything, from fixing a defect to the implementation of a new feature. The change doesn't even have to come from within the boundaries of the organization, changes by external parties can also be the source of a change.

2.2.5 Requirements change risks and project performance

2.2.5.1 Project outcome

According to Ferreira et al (Ferreira et al., 2009), risks of changing, or volatile requirements, are those effects of the change which affects the project outcome, either indirect or directly. This is in compliance with the earlier given definitions of software risks. Ferreira et al. state based on their literature study and survey data that requirements volatility can increase the job size, extend the project duration, cause major rework, affect other project variables such as morale, schedule pressure, productivity and requirement error generation. Nurmuliani et al. (Nurmuliani et al., 2004) state a number of studies which identify that requirements volatility is one of the major problems for the software industry causing cost overrun and other major difficulties during development (Nurmuliani et al., 2004).

2.2.5.2 Project life-cycle

What researchers agree on is that requirement changes can occur at different moments in the life-cycle of a software project (Ferreira et al., 2009). For example, during development it is general accepted that when requirements change they may introduce errors when developers

Characteristic	Description/values
Trigger	The trigger of the change. <i>For example:</i> change to business case, increased customer understanding, new available technology.
Domain	Where the change comes from. <i>For example:</i> market, organization, vision, specification, solution.
Phase	Project phase when change was identified. <i>For example:</i> requirements, design, coding, acceptance test.
Discovery Activity	Activity during which the change was identified. <i>For example:</i> Provide business case, define goals, define requirements, define architecture, specify test, implementation.
Project manager's control	Project manager's control of change identification. Can ordinarily be expressed as very low up to very high.
Stakeholders	Number of stakeholders roles involved agreeing the change.
Cost	Change cost expressed in days.
Value	Business value to the customer. Can ordinarily be expressed as very low up to very high.
Opportunity	Is the change an opportunity or a defect.
Description	Free text.

Table 2.4: *Characteristics of requirement changes according to McGee (McGee, 2011).*

implement the changes. This can happen when the requirements are ambiguous or when developers are uncertain about details of the requirements (Van Gorp and Bosch, 2002). Having stable requirements therefore is very desirable.

2.2.5.3 Design erosion

According to Van Gorp and Bosch (Van Gorp and Bosch, 2002) changing requirements can also lead to design erosion. Design erosion basically means that because of continuous evolution of a system the design erodes; code is re-factored and parts of the architecture redesigned over time because of changing requirements. Van Gorp and Bosch describe four problems causing design erosion (Van Gorp and Bosch, 2002):

- **Traceability of design decisions** Common notations lack the expressiveness to express concepts during design. Design decisions are therefore difficult to trace.
- **Increasing maintenance cost** Complexity of the system may cause developers to take sub-optimal decisions when for example a more optimal decision would be too much effort.
- **Accumulation of design decisions** When design decisions are related to each other, a revised decision can have the consequence that previous decisions, which determine the current system, are affected as well resulting in a system incompatible with the requirements which is very expensive to fix.
- **Iterative methods** When designing an architecture it is the aim to design an architecture which can accommodate change in the future. This may conflict with iterative design approaches (e.g. 'agile' approaches), since requirements evolve during the development in these approaches.

The impact of design erosion, or architectural decay, according to Williams et al. (Williams and Carver, 2010) very much depends on when during the development a change is implemented.

Characteristic	Description/values
Motivation	The motivation of the change. <i>For example:</i> An enhancement (to improve the system), defect (resulting from an error, fault or failure).
Source	The origin of the change. <i>For example:</i> Resource constraint, law/government regulation, policy, stakeholder request
Criticality/importance	Consequence of making the change. <i>For example:</i> Risk, time, cost, safety
Developer experience	Indicates how well the developer(s) implementing the change understand the system architecture. <i>For example:</i> Minimal, localized, extensive
Category	Classifies the type of change. <i>For example:</i> Corrective, perfective, preventative, adaptive
Granular effect	The extend to which the change affects the architecture. <i>For example:</i> Functional/module, subsystem, architectural, system of systems
Properties	If the change is static it affects <i>logical</i> structures of the software, whereas a dynamic change affects the <i>runtime</i> structure.
Features	Determine the impact of the change on functional requirements of the system. <i>For example:</i> Devices, data access, data transfer, system interface, user interface, communication, computation, input/output.
Quality attributes	Areas impacted when the change addresses a software quality attribute. <i>For example:</i> Usability, reliability, functionality, portability, availability, maintainability, scalability, efficiency.

Table 2.5: *Characteristics of requirements changes according to Williams et al. (Williams and Carver, 2010)*

When late in the development cycle, the architectural complexity is likely to be higher, thus the impact or required effort is expected to be higher also (Williams and Carver, 2010).

2.2.6 Conclusion

Definitions from literature of software risks, requirements risks and design erosion are all *static risks* having a negative effect on the outcome of the software project. However, if we look at the definition of a *requirements change*, the goal can also be to enhance parts of the system. In that sense, the risk of some part of the system changing because of a requirements change with the purpose of enhancing, or improving something is a *dynamic* risk. Therefore, we need a broader definition of risks for this study. In this study a requirements change risk is regarded as an event or thing with some uncertainty, where choice is involved and which may have either a positive or negative effect on the outcome of the software project. The source of the change doesn't even have to be within the boundaries of the organization. A risk of a requirement change then can be any mechanism triggered by the change having a positive or negative outcome on the organization or the system. We expect that the risks are very diverse and exists on a business, organizational, technical and people level as we can see from the challenges of working on a shared codebase.

§ 2.3 WHAT IS WORKING ON A SHARED CODEBASE?

2.3.1 Development structures

On what basis development teams are formed determines the structure and responsibilities of the teams. From our study we learned that four different development structures can be distinguished: project-driven, component-driven, product-driven and feature-driven development. We will discuss them below.

2.3.1.1 Project-driven development

Project-driven development is also called resource-pool or matrix management (Larman and Vodde, 2009). In this organizational structure, people work in one or multiple project teams of which the composition may shift over time. People here are considered more a resource and based on their skills, expertise and availability, project teams are formed and assigned a role within a project team. The scope of the project is usually large (months, not weeks). This organizational structure is depicted in Figure 2.1. As can be seen, the actual products and shared components are not shown in the figure, since the relation between them really depends on the project's scope. A project may be defined as developing some product, refactoring a large component, building a set of new features, etc. This kind of structure can usually be seen in the larger organizations.

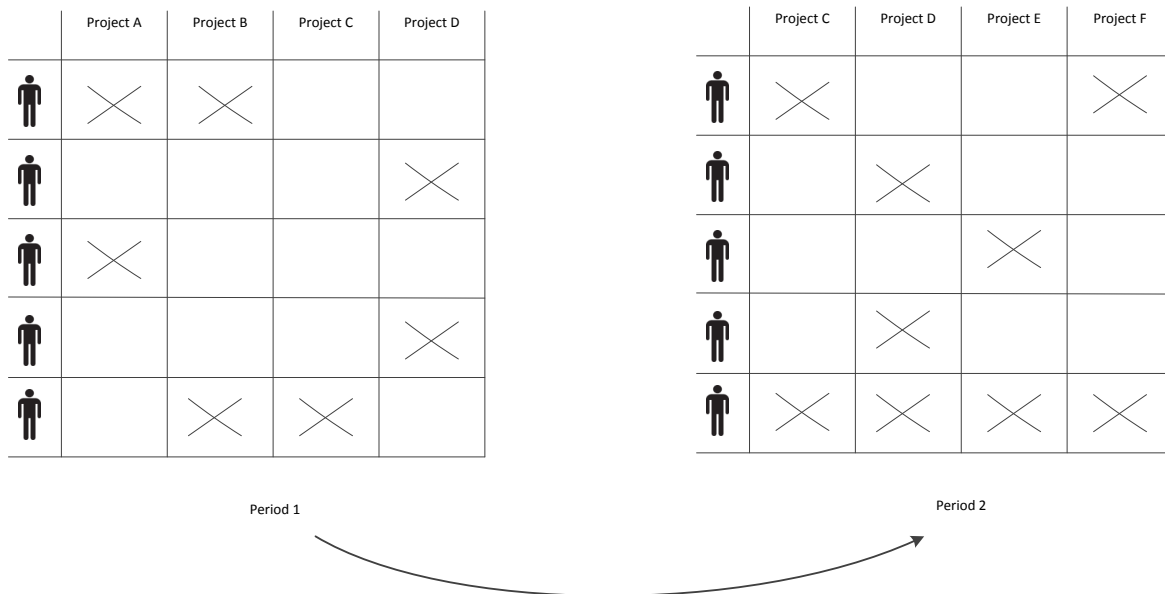


Figure 2.1: *Project-driven development.*

2.3.1.2 Component-driven development

Component-driven development is organized around, as the name implies, components. Teams are given the responsibility and ownership of a component. The work distribution of development of features and products is centralized via the role of managers who keep the overview of the work. The concept of component-driven development is depicted in Figure 2.2. Features for some product may encompass modifications across multiple components, hence the necessity of a coordinator (Van der Linden, 2001).

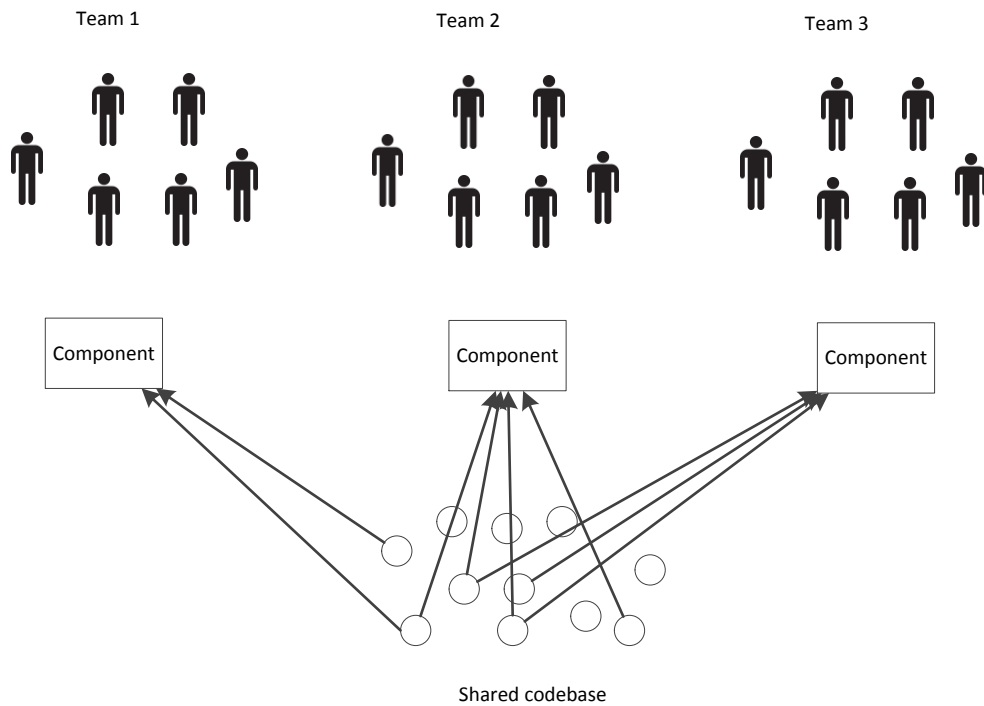


Figure 2.2: *Component-driven development.*

2.3.1.3 Product-driven development

Teams in a product-driven development environment are completely responsible for one product and its features. A product team is often lead by a product owner who is responsible for client acquisition and the overall development of the product. Product teams are responsible for the complete life-cycle of a product, including deployment and maintenance. Of course the responsibilities outside the scope of development may be distributed to the customer or an additional partner. This organizational structure is depicted in Figure 2.3.

2.3.1.4 Feature-driven development

The concept of feature teams is discussed in a book by Larman (Larman and Vodde, 2009), of which we will discuss the contents below.

A feature team is a cross-functional, long-lived team who sequentially conduct the development cycle of many customer features. The team is composed of around 6-8 persons each with there one expertise, but without a permanent role. A feature team can exists of developers, testers, analysts, architects and interaction designers, but the roles are not fixed. Developers can do analysis, analysts can write code.

Feature teams are often compared with the phenomenon Scrum teams from the agile development methodology world. A good Scrum team is a feature team, but a feature team is not a Scrum team. Teams in a feature-development environment have the responsibility of the complete life-cycle of a feature. The scope of a feature can be very broad (implement a number of financial charts) to very small (implement a new pop-up). The team then fulfills all the functions, making the requirements, designing, implementing and testing. A quote from the book of how feature teams in Ericsson are employed:

“The feature is the natural unit of functionality that we develop and deliver to our customers, and his is the ideal task for a team. the feature team is responsible

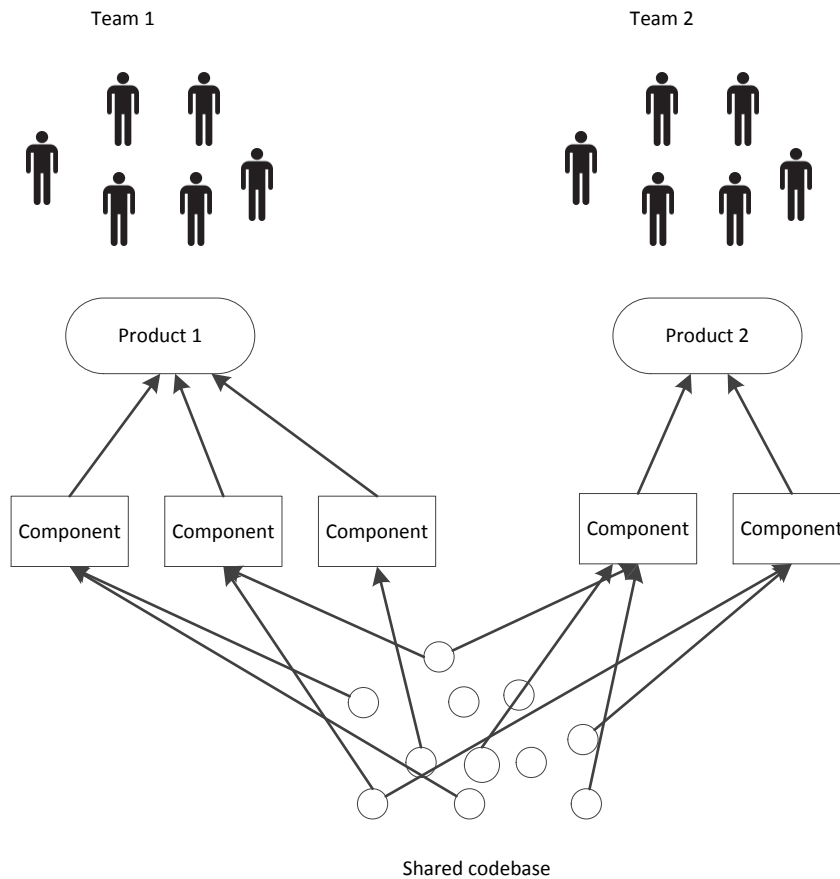


Figure 2.3: *Product-driven development.*

for getting the feature to the customer within a given time, quality and budget. A feature team needs to be cross functional as it needs to cover all phases of the development process from customer contact to system test, as well as all areas [cross component] of the system which is impacted by the feature.”

The effectiveness of feature teams is claimed based on two basic arguments. The first is that feature oriented development encourages learning and personal development. As a team you are constantly deployed in another domain and/or product and as an individual you can switch between roles in your team. This keeps people sharp and trained. Also, this leads to the second argument, which is that features often span a number of components. Over the course of the development cycle, different teams work on the same components, improving and adjusting each other’s code.

2.3.2 Shared codebases challenges

The topic of this study is risks of requirements changes in a shared codebase environment. We explored requirements risks in the context of project performance and in this section we will discuss what working on a shared codebase implies.

2.3.3 Challenges of working with a shared codebase

Ghanam et al. (Ghanam et al., 2012) conducted an ethnographic study among a large number of software teams working with and creating reusable components. According to Ghanam et al.

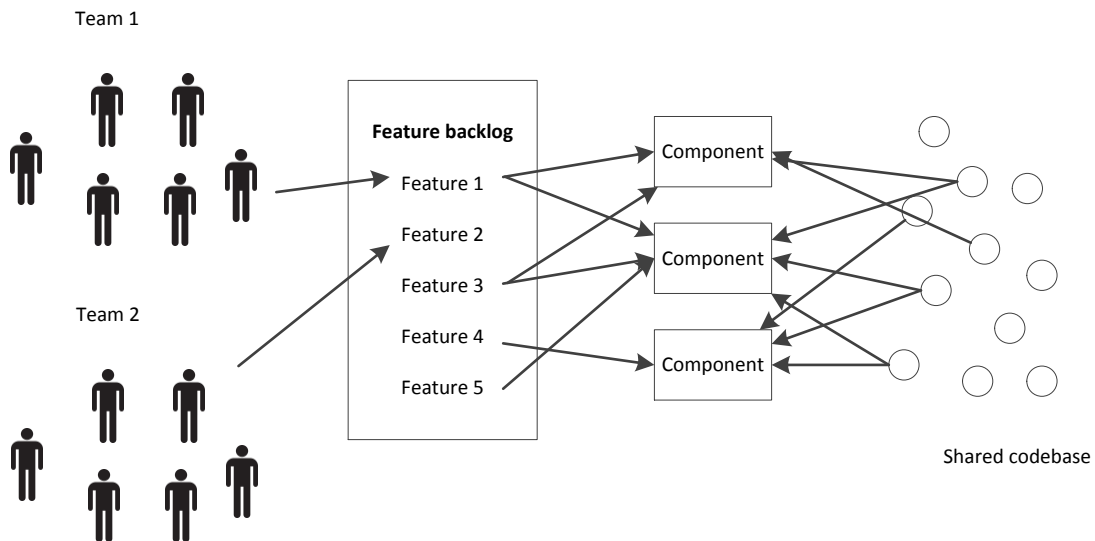


Figure 2.4: *Feature-driven development.*

the following challenges structure exists in these environments:

- **Business challenges**

- **Business strategy** To accommodate new products when the business strategy shifts towards a new market segment, changes may be required to the platform which can affect the support by the platform of other products.
- **Product-driven/platform development**
 - * **Instability** Some components may not be stable enough to be used in the platform and causes problems among products, for example when they are updated frequently or contain a lot of defects.
 - * **Dominance of a mainstream product** When there is one main revenue generating product, the priority of the development effort may be tightly coupled to the need of that product. This may cause the platform to become under-engineered.
 - * **Competing goals** Business goal (for example: fast delivery) may conflict with the adoption road map of the platform by products (technical road map).

- **Organizational challenges**

- **Communication**

- * **Among platform teams** Required to (1) assign responsibilities to components (2) resolving dependencies among components (3) agreeing on protocols and internal interfaces (4) synchronizing releases (5) arranging sharing of resources
- * **Between platform teams and application teams** Product teams for example need to know how to access platforms in order to integrate components
- * **In distributed development** For example: physical separation of teams may lead to communication deficiencies .
- * **Between business units** See silo's.

- **Organizational structure**

- * **Silos** A silo can be seen as the result of an organizational structure where business units or teams act as independent entities having their own local management and no motivation to adhere to a centralized decision-making body or to share information with other units.
 - * **Decision-making** On what organizational level should decisions regarding the platform be made? Centrally or leave it up to the teams?
 - * **Stakeholder involvement** Who to involve in the decision-making process and design of platform components?
- **Agile culture**
- * **Feature vs component teams** Agile development is more focused on teams having end-to-end responsibility of features whereas component based focuses on teams working autonomously on some sub-system. From the research conducted by Ghanam et al. it seemed that a combination of both was needed in order for the subject company to be successful in adopting a platform strategy.
 - * **Team autonomy** The risk of having high autonomous teams together for a long time is that they turn into silos.
 - * **Business-value thinking** When working in an agile environment, companies tend to have a ‘deliver direct value for the business’ perspective. Moving to a platform strategy then has zero visible value for customers, hence the organization has to motivate teams and justify the investment required.
 - * **Product ownership thinking** Teams can be protective of their assets which then makes the transition to a platform difficult.
 - * **Agility versus stability** Agile is focused on accommodating change, which may cause trouble when also wanting a stable platform.
- **Standardization**
- * **Of documents** When documentation standards are not consistent across teams/business units, people are less likely to refer to them
 - * **Of practices** Software reuse requires code conventions and testing standards in order for developers to be willingly to reuse code of others.
 - * **Of tools and technical solutions** Reuse can be difficult if different version control systems or testing platforms are used.
 - * **Of acceptance criteria** When components are proposed to be added to the platform, standardization of acceptance criteria is required.
- **Technical challenges**
- **Commonality and variability**
- * **Reuse** during development developers constant need to look for opportunities to reuse and detect redundancy in the platform.
 - * **Variation sources** Variation required to support by the platform may have different sources; for example business needs or customer needs.
 - * **Cross-cutting concerns** A change may be needed in not all products, but for example a sub-set.
- **Design complexity**
- * **Different actors** Variability can be driven by the business trying to target different market segments.

- * **Requirement of combinations** The requirement from the business to combine components and services to build unique products.
- * **Requirement of maximizing reuse** For example when designing a platform for different hardware platforms, components must be as general in use as possible. Developers must be agnostic to this aspect.
- **Code contribution**
 - * **Accessibility/retrievability** Visibility of assets, either code or documentation.
 - * **Platform quality** When different teams change different parts of a platform on regular basis, quality of the platform may be affected. An audit program can resolve this, but without standardization of acceptance criteria across teams and business units this is difficult.
 - * **Platform stability** Changes, ideally, need to be tested against all products which requires a technical solution where the build is forwarded to all environments.
- **Technical practices**
 - * **Testing** What should be tested in the platform and what in the context of the specific applications? Should a change be tested on all instances of the platform?
 - * **Continuous integration** Updates need to propagate to all environments.
 - * **Release synchronization** Synchronizing releases of different components; what component is supported by what version of other components?
- **People challenges**
 - **Resisting change** Adopting a platform strategy may be difficult when people are not seeing the value of the change, perceiving the change as irrelevant, or having to make adjustments for the new work environment.
 - **Technical competency** People need a specific set of skills to write platform code, because it requires knowledge of reuse patterns and design paradigms.
 - **Domain knowledge** Engineers need to have domain knowledge in order to make decisions regarding what is important for the platform and what for a specific application.

2.3.4 Impact analysis

When a requirement changes, conducting an impact analysis gives insight in the affected assets which is valuable information when making a cost estimation of the work. However, there is a lot of flavors of impact analysis. Below we discuss a classification, depicted in Figure 2.5 hierarchically.

- **Source code**

As can be seen in Figure 2.5 this type has the most available methods according to the classification of Lehnert (Lehnert, 2011). Source code methods can be further classified in the following subtypes:

 - **Call graphs**

Focuses on analyzing source code and extracting method calls between source entities and displaying these calls by means of a graph. This graph can then be used by developers to estimate the propagation of a change.

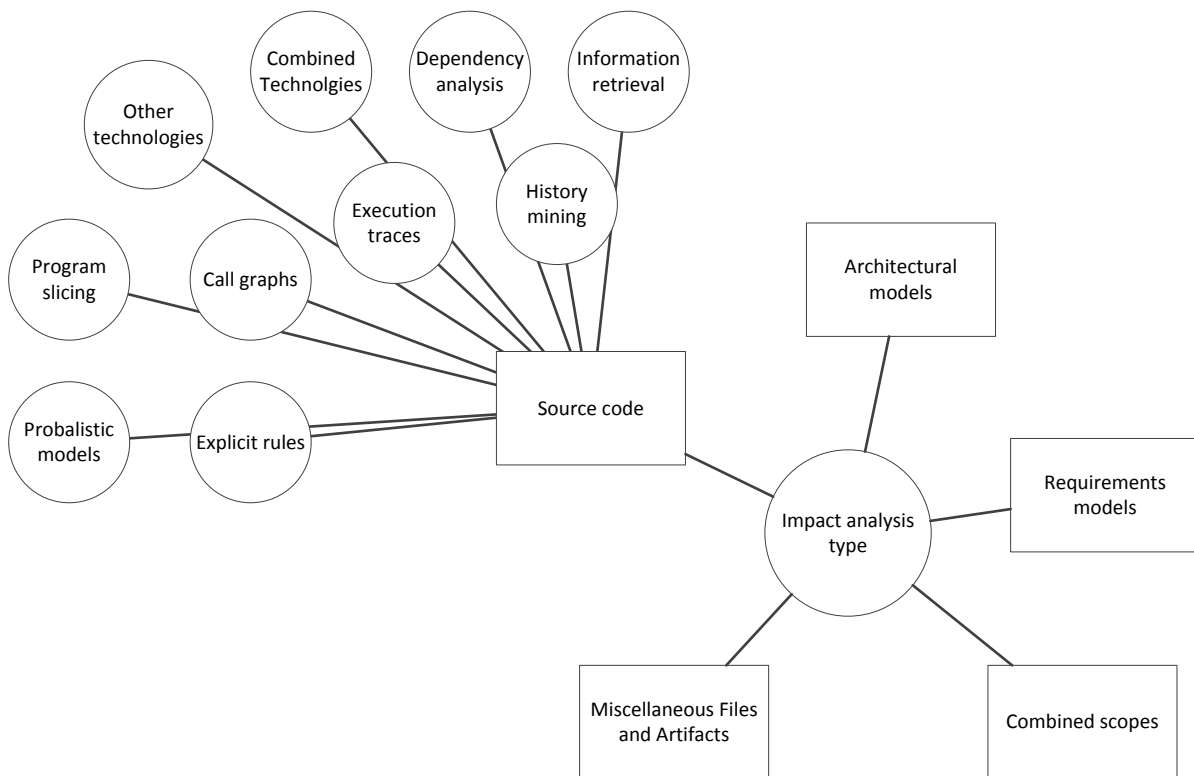


Figure 2.5: *Types of impact analysis methods abstracted (Lehnert, 2011).*

- **Dependency analysis** Different dependencies can exist between source code entities, such as control, data or inheritance dependencies. By means of static source code analysis these methods compose graphs or matrices to display the dependencies.
- **Program slicing** This approach is built upon dependency analysis. With slicing a dependency analysis is conducted where the program code is stripped from all statements which do not comply to the slicing criteria.
- **Execution traces** Dynamic analysis approaches which instead of analyzing static code analyze method calls at run-time and compose call graphs using dynamic execution data.
- **Explicit rules** Techniques which statically analyze code and based on a rule set determine what possible code entities require change. For example, when an interface changes all classes implementing this change need to change as well.
- **Information retrieval** Techniques which focus on the traceability between design documents and code entities and themselves. For example, you can retrieve relevant design documents by extracting meaningful words from design documents and use their matching degree with code entities to prioritize the documents. Another example is calculating the level of change of classes to investigate change patterns in an architecture.
- **Probabilistic models** Models which for example can compute the probability an entity is impacted by a change.
- **History mining** Data mining approaches which use information of changes from the past to predict change impact by for example looking at classes clustered by type of change.

- **Combined technologies** Analysis methods which use a combination of the types described above.
 - **Other technologies** Two methods which cannot be classified according to their used technology.
- **Architectural models**
Sometimes source code is difficult to analysis or not available for the person conducting the analysis. This category encompasses changes among UML models, logical models which can model for example architectural decisions, or visualizations of change history of software components. Also this category focuses on the impact among architectural design documents. Most of the models used in this category requires the user to maintain a certain model during the development to be able to conduct analysis.
 - **Requirements models** These models focus on the change of requirements on the specification level of a software product. Models which can be applied in this context are for example probability models, which can calculate the probability that some requirement is affected by a change in some other requirements. Another example are use case maps, which analyzes the impact of a change based on system scenarios. Also document indexing, which means linking requirements documents to certain keywords depending on its semantic, can help in assessing the impact of a change and prioritizing the impact.
 - **Miscellaneous files and artifacts** An example of this category of analysis is extracting co-changing files from CVS repositories by finding file clustered according their changes. This approach can find non-trivial changes among otherwise non-related code artifacts.
 - **Combined scopes** Endless combinations between the techniques discussed above can be used to perform impact analysis. Examples are reflecting business process changes on source code, tracing requirements to architecture or linking test cases, source code and requirements.

§ 2.4 RESEARCH APPROACH

In this section the research questions are further explained and defined in terms of goals, expected outcomes, the relationship between the questions and where the results can be found.

2.4.1 Research questions

Q1: *What are the risks of a shared codebase environment with respect to changing requirements?*

We now know what a risk is, what a changing requirement is and what the risk of a changing requirement implies. We also explored what working on a shared codebase yields and what challenges are associated with this. What we do not know yet is what the risks are of changing requirements with respect to working on a codebase. While we think that there is a large overlap in issues of working on a shared codebase, we did not find any specific material addressing requirements changes. So the first research question has as goal to find the general risks associated with changing requirements in a shared codebase environment. We think that the best way for this is to analyze industrial case studies for experiences of working with a shared codebase and changing requirements. So for the first research question:

- Derive risks from practice by conducting a literature study of industrial case studies (chapter 3).

Q2: *What mitigation approaches can be used to mitigate these risks?*

The second research question has the goal to analyze the approaches and techniques which can be used to mitigate the risks from the first research question. We think that the best way to find usable approaches is to directly look for best practices used within Topicus. So in order to answer the second research question:

- Interview employees at different units in Topicus about issues, solutions and best practices of working on a shared codebase (chapter 4).

Q3: *What are the relevant risks for FinCare?*

The third research question has the goal to take the risks from the first research question, take the issues from the interviews at the business units of Topicus, study the organization, goals and shared codebase to list potential issues for FinCare. This is done by:

- Conduct a case study at FinCare and find the most relevant risks for FinCare in terms of impact and likelihood of occurrence (chapter 5).

Q4: *What risk mitigation approaches are suitable to implement by FinCare to mitigate the potential risks?*

The final research question has the goal to frame the possible mitigation approaches in the context of FinCare and recommend what approaches would be suitable to adopt by FinCare. This is done by:

- Discuss for the most relevant risks the suitability of the solutions and best practices found during the interviews in the context of FinCare (chapter 6).

2.4.2 Research construct

The research questions, the activities to answer them and the relations between them are illustrated in Figure 2.6.

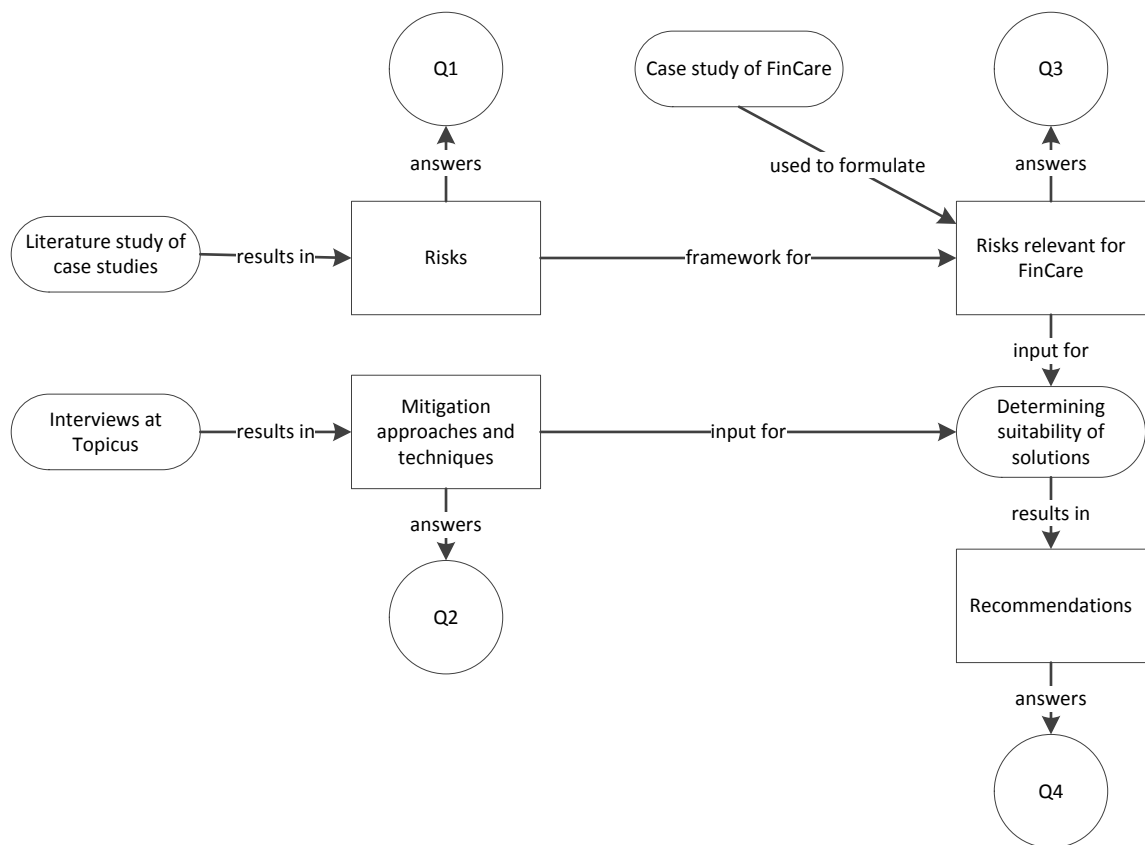


Figure 2.6: *Research construct.*

§ 2.5 CONCLUSION

In this chapter we defined risk of a changing requirement and what yields working on a shared codebase. Based on this, we formulated the research approach for this study.

- Chapter 3 -

Literature study of case studies

§ 3.1 INTRODUCTION

This chapter contains the literature study of risks of shared codebases as described in case studies from literature. In the context of the thesis this literature study is to answer the second research question from a theoretical and industrial perspective: *what are the risks of a shared codebase environment with respect to changing requirements?*

§ 3.2 RESEARCH APPROACH

For the research approach we follow the first 8 steps of the approach of Wolfswinkel et al. (Wolfswinkel et al., 2011) to organize the search for literature and the approach of Webster and Watson. (Webster and Watson, 2002) for identifying relevant concepts. The first 8 steps of Wolfswinkel et al. are: 1.1) define scope, 1.2) identify fields of research, 1.3) find corresponding databases, 1.4) define search terms, 1.5) search, 1.6) filter out doubles, 1.7) cut down sample based on title+abstract, 1.8) cut down sample based on full text. Step 1.9 (forward and backward citations) will only be done if we do not find adequate material in the first place. Since the goal of this literature study is not to have all published material available in the field included, we will only perform this step to find additional material if necessary. Step 1.10 (verify final dataset) is according to Wolfswinkel et al. done in research pairs, but since this is a solo project its a bit unpractical to do this. Verifying the set here will be done by reading the full text and deciding on the relevancy after analyzing its content. Step 2.1 and onward describe a grounded theory approach to analyze the material which is far to rigor for this study. For this we will use the approach of Webster and Watson. (Webster and Watson, 2002). All articles will be read and categorized according to theme and concepts they address in other to find relations among the material.

3.2.1 Define scope

In order to answer the research question, we are looking for case studies describing situations where functionality is shared among multiple applications or projects in the form of a framework, platform or other form of asset reuse. In the case studies we are looking for approaches used, techniques applied, problems described and other relevant experiences from code reuse. Also we are interested, but not exclusively, in how all this works in an agile environment because of the agile culture of Topicus.

3.2.2 Identify fields of research

For the literature study we will look in published material (article and conference papers) available through the Scopus and WebOfScience databases.

3.2.3 Define search terms

From the goal of the literature study we abstract the following key topics: ‘code reuse’, ‘software reuse’, ‘shared codebase’, ‘framework’ and ‘repository’. For all of these key topics we want a) case studies in b) a collaborative setting in c) optionally an agile development environment. We decided to split the search for articles in two distinct queries, one focusing on software reuse in an agile or/and product line engineering setting and the other on shared repositories regardless of development methodology. As for the knowledge domains, we both looked in the Computer Science and Management Science domain in the period of 2000-2012 to find recent material. We are not looking for fundamental theory, but recent industrial experiences.

3.2.4 Search

The resulting queries and their results are:

```
1 TITLE-ABS-KEY(("Code reuse" OR "software reuse" OR "collaborative develop*" OR "
  framework develop*" OR "product line engineering" OR "shared codebase") AND ("
  case study" OR "agile")) AND PUBYEAR > 1999 AND (LIMIT-TO(SUBJAREA, "COMP") OR
  LIMIT-TO(SUBJAREA, "BUSI"))
```

Listing 3.1: *Reuse in PLE/agile context query in Scopus: 384 results*

```
1 TS=(("Code reuse" OR "software reuse" OR "collaborative develop*" OR "framework
  develop*" OR "product line engineering" OR "shared codebase") AND ("case study
  " OR "agile")) AND SU=( "COMPUTER SCIENCE" OR "OPERATIONS RESEARCH MANAGEMENT
  SCIENCE" ) AND PY=( 2001 OR 2002 OR 2003 OR 2004 OR 2005 OR 2006 OR 2007 OR
  2008 OR 2009 OR 2010 OR 2011 OR 2000 OR 2012 )
```

Listing 3.2: *Reuse in PLE/agile context query in WoS: 206 results*

```
1 TITLE-ABS-KEY(("repositor*" OR "codebase*" OR "version*" OR "revision*") AND ("
  shar*" OR "distribut*" OR "collaborat*") AND ("case study")) AND PUBYEAR >
  1999 AND (LIMIT-TO(SUBJAREA, "COMP") OR LIMIT-TO(SUBJAREA, "BUSI"))
```

Listing 3.3: *Repository query in Scopus: 381 results*

```
1 TS=(("repositor*" OR "codebase*" OR "version*" OR "revision*") AND ("shar*" OR "
  distribut*" OR "collaborat*") AND ("case study")) AND SU=( "COMPUTER SCIENCE"
  OR "OPERATIONS RESEARCH MANAGEMENT SCIENCE" ) AND PY=( 2001 OR 2002 OR 2003 OR
  2004 OR 2005 OR 2006 OR 2007 OR 2008 OR 2009 OR 2010 OR 2011 OR 2000 OR 2012
  )
```

Listing 3.4: *Repository query in WoS: 459 results*

3.2.5 Filter out doubles

Merged together and filtered for doubles the search yields 356 results in total. Merging and removing doubles was done using Microsoft Excel. Doubles were removed based on title.

3.2.6 Cut down sample based on title+abstract

In order to cut down the sample we read the titles and abstract and decided the paper to be relevant for this literature study if the study is in industrial study where either:

1. Experiences with some applied technique to counter stated problems/issues with reuse/-platform development are described
2. or general experiences with code reuse or shared codebases are described

Applying this heuristic was done in 2 iterations and resulted in 80 papers.

3.2.7 Cut down sample based on full text

A sample-set of 80 papers is a too large for the purpose of this study, so we scanned all 80 papers briefly in full text to judge its relevancy. This resulted in a set of 30 papers. Papers rejected here were full example poster papers, short papers, describing small-tools or prototypes in a merely academic setting (e.g. not a proper industrial case study). The resulting set yields 30 papers which were then read in detail and analyzed. After reading the full text, 4 more paper were dropped because of irrelevant content for this literature study and 3 more because of duplicate content.

§ 3.3 RESULTS

The resulting papers read for this study provided very useful information, but we did not find detailed case studies about working with a shared repository among different project teams. That is, not in these terms. From reading the articles we noticed that authors use terms like ‘domain engineering’ or ‘shared software assets’ to describe what is essentially working on or with the same source code, code base, or repository. Nevertheless, all articles discussed below are industrial case studies which we categorized according to their focal topic:

- Benefits and misfits of product line engineering (PLE) (Mohagheghi and Conradi, 2008)
- Combining agile and PLE (Hanssen, 2011; Hanssen and Fægri, 2008; Noor et al., 2008)
- Using a reuse strategy (Ghanam et al., 2012; Otsuka et al., 2011; van Gurp et al., 2010; Eaddy et al., 2008; Chapman and van der Merwe, 2008; Rothenberger, 2003)
- General development issues: changes (Schröter et al., 2012; Ramasubbu and Balan, 2010; Gupta et al., 2010; Slyngstad et al., 2008; Perry et al., 2001), collaboration and communication (Beckhaus et al., 2010; Ponte et al., 2008; Munkvold et al., 2006)

We will not discuss the results according to those themes, but according to the concepts they address in relation with changing requirements in terms of risks. The concepts were collected iteratively while reading the articles using the method of Webster and Watson (Webster and Watson, 2002). The following concept were found:

Concepts: *Volatile market, client influence, market pressure, short-term versus long-term, ambiguity, scope, scattered functionality, development of reusable components, communication and knowledge sharing, experience in reuse, adoption of a PLE approach, and agile and PLE.*

In the following sections we will discuss each concept and their relationship with changes and associated risks.

3.3.1 Volatile market

External (outside of the company’s environment) events can be a cause of changes in requirements (Mohagheghi and Conradi, 2008). Examples of these external events are unpredictable market conditions or customer demands. In a study by Mohagheghi and Conradi (Mohagheghi and Conradi, 2008) it is shown that if requirements change stem from those events, application-specific components would be more affected than other components. This is most likely because most external events stem from some specific customer instead for the whole market.

When the market operated in often has to deal with these kind of events, it is called a volatile market. Operating in such a market can be a risk since in these markets it is of strategic importance to be able to respond quickly to changing market conditions. When working with a framework, concessions then have to be made. Developing reusable components now may be of benefit later, but inhibits the ability to respond quickly to market changes. In a case study by Hanssen (Hanssen, 2011) this has been dealt with by having several and frequent links to customers. This allowed the company to quickly deduce market changes and adapt their strategy accordingly. However, if these predictions are inaccurate, a highly reuse focused approach can cause the company ‘enforcing wasteful effort related to adherence to reference architectures, organizational overhead and reuse strategies’ (Hanssen and Fægri, 2008).

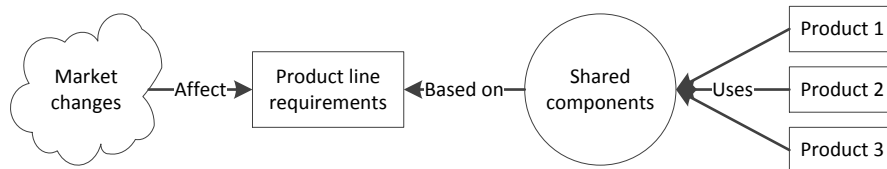


Figure 3.1: *Volatile market.*

3.3.2 Client influence

Having a short rope with your customers thus can be beneficial, but client influence is also a risk. In a study by Rothenberger (Rothenberger, 2003) it is shown that when the perceived value of reuse by a customer is low, they have a higher chance to disagree with further investments and planning of reuse activities. According to the authors this perceived value mainly depends on ‘the effort that is invested up front in education about reuse benefits’ (Rothenberger, 2003). Also, when a client’s budget and time constraints are directly reflected upon developers, lower reuse rates were achieved (Rothenberger, 2003). In the study it was also shown that when the system under development was of low operational importance for the client, it felt that it should not affect other more critical projects. Hence, reuse or framework dependencies were not favored. This is called ‘fear of interconnectivity’ (Rothenberger, 2003). When involving clients in the development process, you therefore need to think about who to involve in the decision-making process of your platform design (Ghanam et al., 2012).

3.3.3 Market pressure

When incorporating a reuse approach, market pressure influences and defines risks for companies. For example, in a study by Ramasubbu and Balan (Ramasubbu and Balan, 2010), development of product features having direct competition was significantly lower compared to development of unique features to the company. The authors conclude that development teams were ‘particularly sensitive to competitive pressures and wanted to stay ahead in the market by uniquely positioning

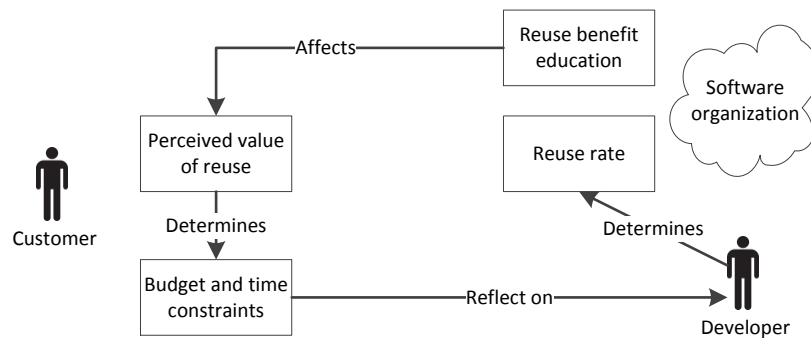


Figure 3.2: *Client influence.*

the product’ (Ramasubbu and Balan, 2010). This pressure can be considered time-to-market pressure, which can come either from within or outside the company.

Market pressure also may, according to the study by Gupta et al., lead to many perfective changes later in the development cycle because of refactoring needs. In this case study this was mainly because rules and business logic were complex and hard to maintain and hence needed refactoring (Gupta et al., 2010).

This can also be seen when relying on standards for your product. Market pressure then can force redesign if a standard change or forced adoption renders some feature incompatible, as shown in a case study by Ramasubbu and Balan (Ramasubbu and Balan, 2010). Market pressure can also be caused by political aspects, as stipulated by Ponte et al. (Ponte et al., 2008) where in a case study the design phase of reusable components was led by political aspects instead of technical aspects.

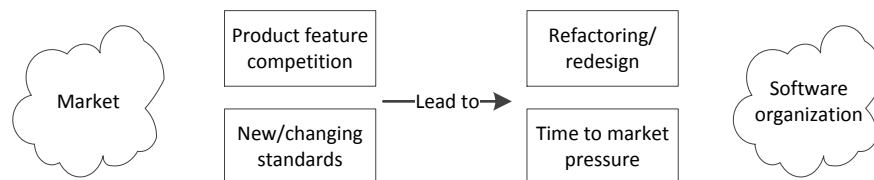


Figure 3.3: *Market pressure.*

3.3.4 Business strategy

As stipulated shortly with respect to a volatile market, planning and time-to-market pressure from clients can be a risk for companies if it compromises their ability to deliver stable products in the future (Hanssen, 2011). When developing products using a shared framework thinking ahead of what functionality should be in this framework and what not can save time and resources later. However, investing up-front is not always desirable if clients are pressuring for fast releases. This is the dilemma of short-term versus long-term goals. In both situations change is dealt with differently. In a short-term planning scenario, changes have to be incorporated in existing deployed parts of the framework which can cause errors or conflicts in already deployed products. However, companies can also choose to deploy different versions of frameworks to deal with this. This of course leads to maintenance problems if more versions are maintained in parallel.

Choosing the right course of action for your shared asset development depends on the products you plan to deliver. When you have a lot of different clients in some product line with their own wishes then the number of variation sources for shared assets increases. The more

actors involved, the higher the number of changes can be expected if all actors can influence the development of the shared assets (Ghanam et al., 2012).

When designing or refactoring components for reuse, the development at some point needs to be initiated. In a study by Ghanam it is shown that when one big client is involved, the reusable assets may be developed based on this big client. This is called prioritizing on a mainstream product. This is a risk if other customer's implementations suffer from the design decisions from this mainstream product (Ghanam et al., 2012).

A change of business strategy, for example to serve new markets, new customers is also a risk if this requires a different vision on the shared assets (Ghanam et al., 2012). For example, if at one point a long-term reuse strategy was employed, but later the company switches to a more short-term planning, you have the risk of using a immature component for reuse. Short-term versus long-term planning, agility versus stability or business-value thinking versus customer-value thinking are all very prominent points of debate in the development of reuse assets.

The client in this dilemma is considered an external factor, but internal factors also play a role here. For example, a company may have a business philosophy of focusing on short-term goals preventing a company from adoption reuse in their projects (Sherif and Vinze, 2003). Also, there can be the issue of product ownership thinking when development teams are protective of their assets when moving to a reuse scenario (Ghanam et al., 2012).

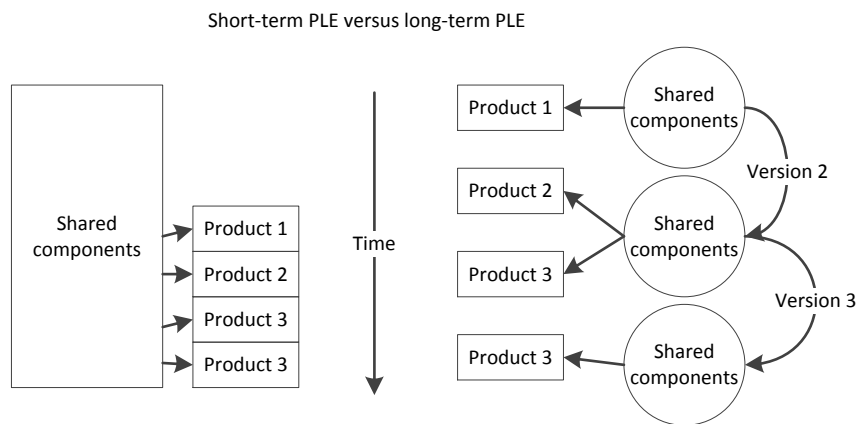


Figure 3.4: *Long-term versus short-term PLE.*

3.3.5 Ambiguity

Developing reusable components requires clear requirements. If different parties want to use some component, requirements from all parties need to be combined into a technical design which satisfies all needs. Besides the fact that this may require concessions from all parties, ambiguity or unclear requirements may cause errors in the implementation which then may cause defects in products relying on this component. In a study by Gupta et al. it is shown that unclear requirements in their case led to many perfective changes later in the development cycle (Gupta et al., 2010).

Besides errors in the implementation, ambiguity in artifacts may cause different interpretations of the contents of the artifacts among actors. This may cause deviations in development of components over time, as shown in a case study by Ponte et al. (Ponte et al., 2008). More generally, ambiguity of goals between actors can lead to standardization disputes when artifacts become the center of collaboration between actors. This problem can also exist internally if for example different interpretations or goals exist of for example a planning or road-map (Ghanam

et al., 2012).

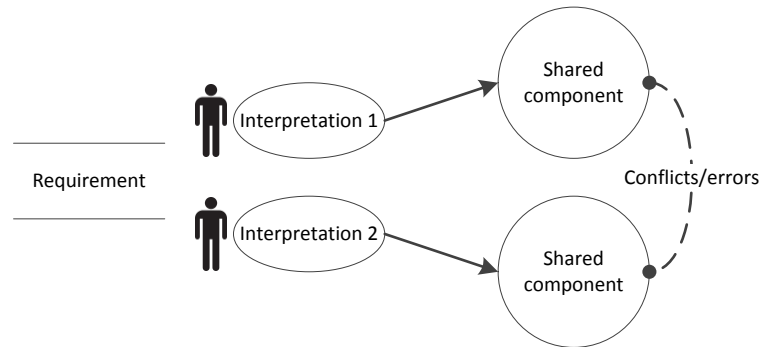


Figure 3.5: *Asset ambiguity.*

3.3.6 Scope

When setting up components for reuse, they are often designed with some product scope in mind. Widening this scope then can become a risk. In a case study performed at Fujitsu where reusable components were developed, it was observed that ‘the scope of core asset creation should not be excessively expanded before the product line development is sufficiently mature’ (Otsuka et al., 2011). A product line approach here was adopted because of the high expected development costs and the necessity to have reusable components for future products.

This problem was also shown in a case study by van Gurp et al. (van Gurp et al., 2010) where ‘[because of scope widening], the platform and target products increase, the platform has to support more and more potentially conflicting requirements’. In this study it is also shown that besides widening the scope, decisions encompassing a system wide scope also poses risks. When difficult design decisions in the product line platform are made addressing issues of multiple current products, later development of products may be compromised because for them an alternative solution would be better (van Gurp et al., 2010).

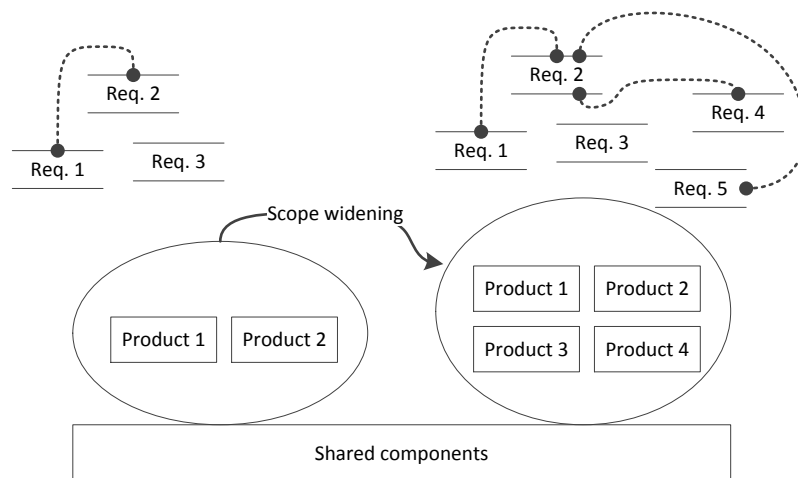


Figure 3.6: *Scope widening and requirement complexity increase.*

3.3.7 Scattered functionality

Development of reusable components may encompass technical risks. In a study by Eaddy et al. it was found that scattered functionality because of inheritance can cause problems for developers when developing shared assets (Eaddy et al., 2008). For developers it may be difficult to oversee all impact of changes in the inheritance tree when it is unclear what other products may use this functionality.

Also, scattered functionality is difficult to deal with if changes are required at the product level for some functionality (e.g., not implemented as a reusable component). Testing costs then may increase if a platform-wide testing has to be executed (van Gurp et al., 2010). In general, deploying ‘differentiating platform features’ is a risk because of this, if they depend on components which require platform-wide testing when they are changed (van Gurp et al., 2010).

Of course there may be very legit reasons for scattering functionality, but for developers it is often difficult to understand the plan behind this if plans are developed delocalized. This may in a worse case scenario result in several incorrect modifications across the system (Eaddy et al., 2008).

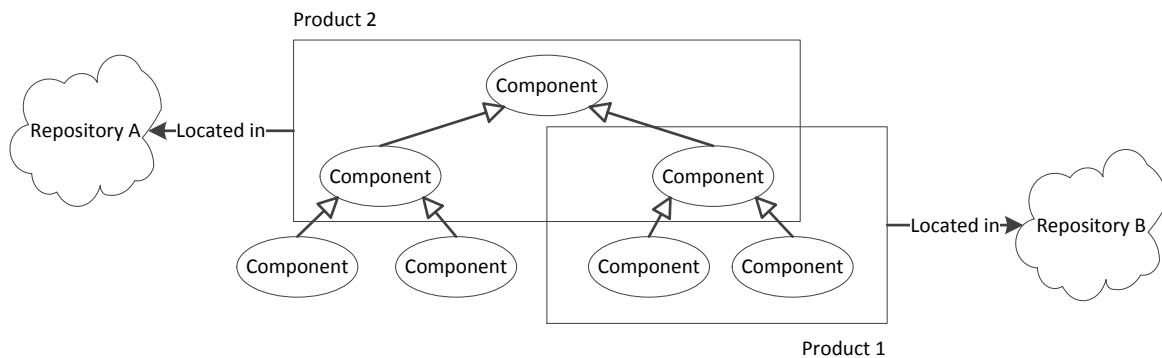


Figure 3.7: *Scattered and delocalized functionality.*

3.3.8 Development of reusable components

On a low technical level working with multiple people on the same artifacts yields two types of problems: semantic conflicts and logical completeness. When working on the same files, you need to merge your changes with the changes of others. This is called a semantic conflict. When synchronizing a consistent build of a system you have to worry about dependencies shared across multiple components in the system, which is called logical completeness. (Perry et al., 2001). Merging or synchronizing files these days is not so much a technical concern; the first problem is often dealt with by incorporating a version control system and the second problem can also mostly be automated by build servers. However, on a functional level the challenge here is to be able to foresee the effect of a merge or synchronization of shared components across products.

In a study of build-level components a number of issues with reusable components on implementation level are discussed. Issues stated here are: component granularity, component coupling, circular dependencies, non-standardized configuration interfaces, early binding of build-level dependencies, single build process definitions, configuration management for component deployment and making a system composition by hand (de Jonge, 2005).

Editing a component which is shared among products may cause multiple files to change can increase ‘code churn’ (Eaddy et al., 2008). When components span multiple products, editing them yields a ‘crosscutting concern’. Having such concerns is a risk if they are rarely documented. Without proper documentation top-down knowledge questions (like where are all

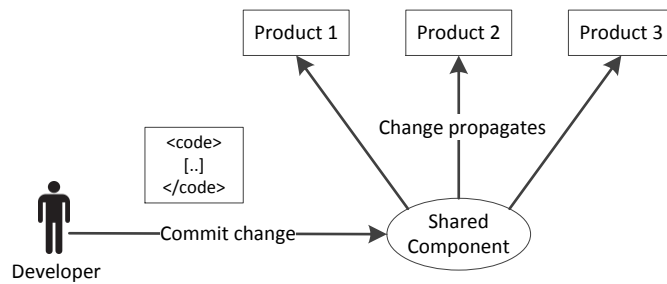


Figure 3.8: *Change propagation.*

the locations where function X is used) and bottom-up knowledge questions (like: what is this piece of code for) (Eaddy et al., 2008) cannot be answered easily.

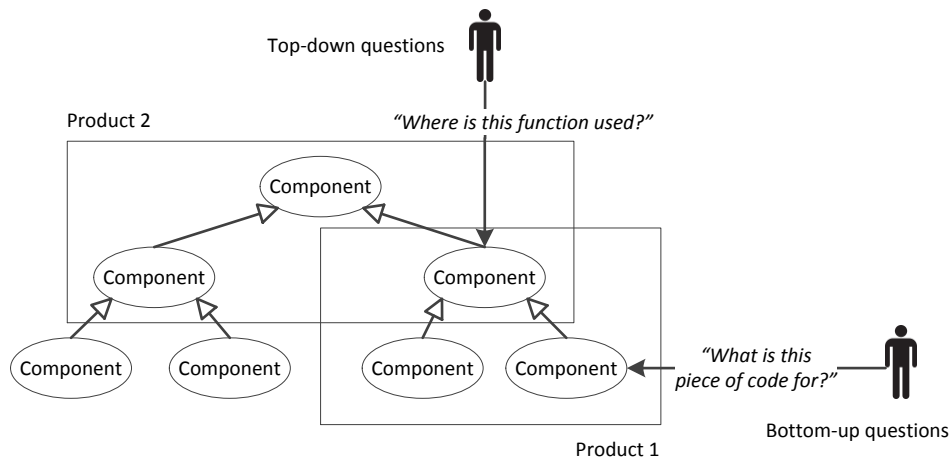


Figure 3.9: *Knowledge questions.*

Iteratively working on shared components is shown to be both beneficial and a risk. In a study by Mohagheghi and Conradi (Mohagheghi and Conradi, 2008) it is shown that reuse results in a lower fault density and code modification rate of those components over time. Also the authors show that reused components had ‘significantly fewer post-delivery faults than non-reused ones’ (Mohagheghi and Conradi, 2008).

Ramasubbu and Balan (Ramasubbu and Balan, 2010) show in a case study that a company may use different strategies for different natures of change: incremental changes, modularity changes, architectural changes, or radical changes. The company prioritized their attention according to the degree the changes affected modularity and architectural changes (Ramasubbu and Balan, 2010).

In general four classes of changes are used in software literature: corrective, adaptive, perfective and preventive (Slyngstad et al., 2008). The type of change done on shared functionality can be an indication of the risks involved. For example, according to Slyngstad et al. (Slyngstad et al., 2008), a ‘scope change’ is related to perfective, adaptive and preventive changes whereas an ‘incident’ relates to corrective changes. Scope changes then may require a more rigor and coordinated effort between development teams whereas incidents are more locally handled. In combination with test-driven development in this case study a defect density decrease of 36 percent per release was achieved (Slyngstad et al., 2008).

3.3.9 Communication and knowledge sharing

Developing components which are used among multiple products requires coordination, communication and knowledge sharing. Different collaboration patterns encompasses different risks in a shared codebase environment.

In a study by Beckhaus et al. (Beckhaus et al., 2010) it is stated that when a collaborative process is heterogeneous, a process has not yet been established and in the case study a negative influence on performance then was observed. It is in general assumed that when knowledge workers frequently exchange information and consult co-workers, processes are more efficient. In this case study it was shown that in most cases of highly centralized collaboration networks the ‘most central employees happen to execute the dispatcher role’ (Beckhaus et al., 2010). In the case of issue tracking, this kind of centralization may introduce a bottleneck ‘since relying on a key user for information exchange makes the entire issue tracking efficiency dependent on his or her availability’ (Beckhaus et al., 2010).

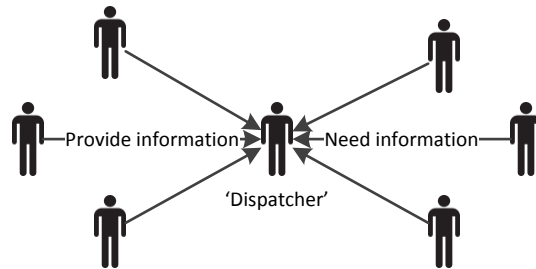


Figure 3.10: *Dispatcher role.*

When collaborative processes are more formal, assets as for example framework road-maps can be used reliably. In a more formal environment they can be expected to be updated often and, when shared, can communicate the status of development for everyone in the organization, even for customers (Hanssen, 2011). This highly depends on the implemented development methodology, but also on the culture of the organization. If the company values direct face-to-face communication over documentation, a more formal approach may not be favored despite the benefits (Hanssen, 2011).

Also, incorporating a ‘reuse champion’ or in general the degree of promotion/emphasis of reuse by a product/team leader affects the attitude of employees towards reusing functionality. Giving trainings or incentives for reuse may influence the successful use of shared components as well (Rothenberger, 2003). Having few incentives for projects to collaborate in the process of tools selection or training schedules can make asset sharing between projects impractical (Chapman and van der Merwe, 2008).

An organization structured in a project-centric way often is described as developing independent one-offs according to the distinctive requirements of a customer. When working in the context of a framework with reusable components or a product family, the development is more focused around a certain degree of commonality stimulating software reuse. In a ‘project-centric’ company development of reusable assets often must be the side-product of the project, reducing the incentives for a project to collaborate in a reuse effort among projects (Chapman and van der Merwe, 2008).

As discussed before, having poorly documented crosscutting concerns makes it difficult for employees to answer top-down and bottom-up knowledge questions. Also, without communication of plans among teams or within team developers may have difficulty understanding these plans coherently which may cause incorrect modifications.

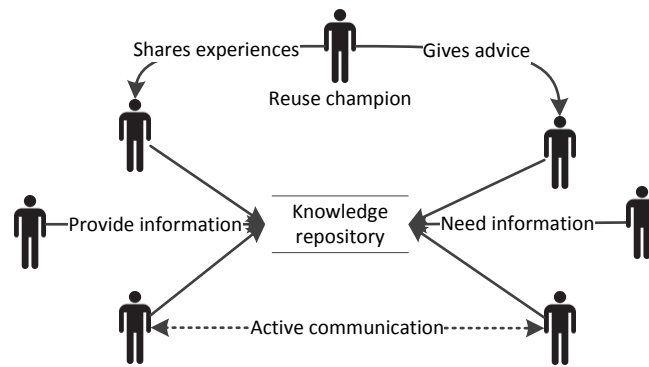


Figure 3.11: *Knowledge sharing and reuse champion.*

When development of a common platform is coordinated by some leading actor, the loss of such an actor may cause actors in the network to go their own way (network unbundling) (Ponte et al., 2008).

3.3.10 Experience in reuse

At the end of the day developers will write code, business analysts will produce documents and team leaders have to distribute resources. The experience people have with these activities in a shared codebase environment therefore is an important factor since it is one of the factors determining the quality of the work. For a developer experience with reuse can be seen in the appliance of reuse patterns, the understanding of the company’s reuse model, the knowledge of component availability/capability (Rothenberger, 2003; Ghanam et al., 2012) and domain knowledge in order to make decisions regarding what is important for the platform and what for a specific application (Ghanam et al., 2012).

In a study by Sherif and Vinze (Sherif and Vinze, 2003) it is shown that having a reuse champion (e.g. an experienced and visionary employee) may be of benefit. The main association with benefits here is that a reuse champion can reduce the impact of having a reuse philosophy on barriers to incorporate reuse. Also in this context, ‘active communication between asset creators and asset utilizers’ and education and training proves to be of benefit (Sherif and Vinze, 2003).

3.3.11 Adoption a PLE approach

A product-line engineering (PLE) approach is something which has to grow within a company. A company may adopt such an approach because of various reasons and implement it in a infinite amount of ways. Also, every company is in a different situation when adopting a PLE approach, making every case even more unique. In some cases PLE may reduce costs and improve quality, but ‘rather only for those few projects visible at the beginning of the project’ (Otsuka et al., 2011).

A *proactive* PLE approach requires upfront investments in a product line architecture on an organizational level to do planning and scoping activities. The tension here is that a predefined architecture puts constraints on the products an architecture can support. In a study by Tang et al. it is stated that this requires developers to ‘shift from short-term thinking to think about long-term benefits’ (Tang et al., 2010). In this case developers indicated that developers often don’t like to be constrained. Also in this study it is stated that this risk can be mitigated by communication the architectural design earlier with customers and developers to gain trust and buy-in (Tang et al., 2010).

A *reactive* PLE approach then is able to deal more easily with changes from the market and from within the business, but has the risks associated with a more ‘agile’ approach and PLE.

Moving from a situation with zero reuse to a situation where a reuse strategy is incorporated is one thing, but in any case a situation where existing components need to be refactored or gradually be made accessible for sharing is very likely. In a case study by Gupta et al. is shown that making some components reusable may reduce the volatility of components, but may also increase the amount of adaptive and preventive changes (Gupta et al., 2010).

In a study by Breivold et al. (Breivold et al., 2008) a number of recommendations are given when moving to a PLE approach:

1. Improve risk management through constant progress measuring
2. Product managers for different products using the product line architecture should synchronize needs
3. Define roles, responsibilities and ways to share technology assets
4. Perform the migration to product lines through incremental transitions
5. Ensure communication between technology core team and implementation team
6. Use tool support for dependency analysis
7. Use architecture documentation to improve architectural integrity and consistency
8. Carefully define variation points and realization mechanisms
9. Use the described method iteratively to handle software evolution

§ 3.4 DERIVED RISKS

From the above sections we can derive a number of risks which we listed in Table 3.1 and explain them in simple words in section 3.4.1 and onwards.

	Risk	References
1	Risk of operating in a volatile market	(Mohagheghi and Conradi, 2008; Hanssen and Fægri, 2008; Hanssen, 2011)
2	Risk of stakeholder influence	(Rothenberger, 2003; Ghanam et al., 2012)
3	Risk of time-to-market pressure	(Ramasubbu and Balan, 2010; Gupta et al., 2010)
4	Risk of evolving standards	(Ramasubbu and Balan, 2010)
5	Risk of political aspects	(Ponte et al., 2008)
6	Risk of business philosophy focusing on short-term goals	(Hanssen, 2011; Sherif and Vinze, 2003)
7	Risk of business value thinking	(Tang et al., 2010)
8	Risk of prioritizing on mainstream product	(Ghanam et al., 2012)
9	Risk of changing the business strategy	(Ghanam et al., 2012)
10	Risk of reusing immature components	(Gupta et al., 2010; Tang et al., 2010)
11	Risk of unclear requirements	(Gupta et al., 2010)
12	Risk of different interpretations of artifacts	(Ponte et al., 2008; Ghanam et al., 2012)
13	Risk of goal ambiguity	(Ghanam et al., 2012)
14	Risk of scope widening	(Otsuka et al., 2011; van Gorp et al., 2010)
15	Risk of scattered functionality	(Eaddy et al., 2008; van Gorp et al., 2010)
16	Risk of delocalized plans	(Eaddy et al., 2008)
17	Risk of iteratively changing reuse components	(Mohagheghi and Conradi, 2008)
18	Risk of changes in product line assets at the product level	(van Gorp et al., 2010)
19	Risk of enhancement to a cross-cutting concern	(Eaddy et al., 2008)
20	Risk of component granularity	(de Jonge, 2005)
21	Risk of circular dependencies	(de Jonge, 2005)
22	Risk of non-standardized configuration interfaces	(de Jonge, 2005)
23	Risk of early binding of build-level dependencies	(de Jonge, 2005)
24	Risk of making a composition by hand	(de Jonge, 2005)
25	Risk of making an application/component reusable	(Slyngstad et al., 2008; Ramasubbu and Balan, 2010)
26	Risk of heterogeneous communication	(Beckhaus et al., 2010; Hanssen, 2011)
27	Risk of centralization in group based collaboration networks	(Beckhaus et al., 2010)
28	Risk of reuse experience level	(Rothenberger, 2003; Ghanam et al., 2012; Sherif and Vinze, 2003)

Table 3.1: *Derived risks from industrial case studies*

3.4.1 Risk of operating in a volatile market

We speak of a volatile market when a software development company has to update their product platform to move along with the latest technological developments in order to maintain her market position. For example, let's assume a company develops a social media platform for a diverse customer base. In the domain of social media, technology develops rapidly so existing technologies are updated often. To stay in business, the company always has to look around for the latest technology developments to anticipate client's wishes and market trends. This however also means that their platform and all depending products often are exposed to change.

Possible indicators for this risk are:

- In the organization there are different applications with customers in a different sub-domain, all based on a shared platform
- Product specific assets are depending on shared components
- Market changes can occur frequent and ripple fast through the organization

3.4.2 Risk of stakeholder influence

Stakeholders in the context of software development are defined as those actors with an interest in the development of the software. Think of customers, suppliers, users and also developers. We speak of stakeholder influence if for example a customer can directly contact a developer to pass on a small feature request. If this feature requires changes on a software platform, this can result in unexpected changes on other products. Stakeholders also can have influence by means of budgetary restrictions what platform activities, which also are relevant for other customers. Possible indicators for this risk are:

- Frequent, direct, informal customer contact
- A relationship with the customer where additional work can be discussed without any formal process

3.4.3 Risk of time-to-market pressure

Let's assume some component can be included in a platform and become available for all other products, but this requires 2 weeks of additional development. The alternative is that the component is not made generic and is directly released and available for one product, because it is promised to the customer right now. This is an example of time-to-market pressure.

Possible indicators for this risk are:

- Fixed priced contracts where for the next number of months the features to deliver are fixed.
- Short-release cycles
- High level of market competition
- A busy road-map

3.4.4 Risk of evolving standards

If standards of protocols or data formats are being used in a platform and such a standard changes, the impact can be different for each product. This can for example be the case for a platform of which the customer of one of the products decides to use a different encoding. If this functionality only is implemented in a generic component, this can mean that a specific component is to be written for this specific customer.

Possible indicators for this risk are:

- A large number of common or shared web-services where for data-communication some data-standard is used
- Incorporated standards where external parties make decisions regarding the contents of the standard

3.4.5 Risk of political aspects

Agreements between consortia made on a political level can influence the development of components if these consortia make agreements on what standards or interfaces between systems are required to be used. Or when they deviate from previous decisions.

Possible indicators for this risk are:

- Standards where external parties have more saying than you
- Consortia involved in the development of components which are part of the system

3.4.6 Risk of business philosophy focusing on short-term goals

With the development of a platform the design and implementation can be planned far ahead in the future if you for example already know what products you expect to roll-out in the coming 2 or more years. A long-term vision means you have to invest a considerable amount of time prior to releasing your software products. A short-term vision focuses more on being able to release live products faster and distribute the development of the platform over a longer period of time.

Possible indicators for this risk are:

- Short-release cycles
- Road-maps with a short scope
- Fixed-priced contracts
- Concessions are done with respect to platform design to achieve early product release

3.4.7 Risk of business value thinking

A strong mind-set of business value thinking is characterized by to what extend the direct visibility of business value in terms of time and money is used in development decisions. For example, let's assume a software company faces the choice to serve a big customer right now with a half-finished platform or continue developing the platform to make it more mature and more stable. The first options generates clear, direct business value while the business value of the latter is less certain.

Possible indicators for this risk are:

- Getting a customer now is favored over taking the risk of investing more first and then roll-out the products
- Rather make the next deadline, and the next and the next than refactoring/improving now

3.4.8 Risk of prioritizing of mainstream product

A mainstream product typically is a large, cash-cow product. We speak of prioritizing of a mainstream product if for example during design or planning activities concessions are made beneficial for this mainstream product, but which may have a negative result on other products. This can happen if stakes are high. Think of concessions like resource allocations or not including some functionality in the scope of other, less important, products.

Possible indicators for this risk are:

- There is a mainstream product to distinguish with relative high financial stakes
- Decisions about resource distribution are based on the mainstreamness of products

3.4.9 Risk of changing the business strategy

When during implementation of a new business strategy it is decided that some platform is being rolled-out in a new market, this can have a significant impact on existing products. For example when emerging requirements from the market are not yet supported by the platform.

Possible indicators for this risk are:

- Unstable market position
- Highly depending on external parties for technology or propositions

3.4.10 Risk of reusing immature components

A component which is still actively in development and which has not proved itself in practice we call an immature component. Already (re)using these kind of components in production as part of a software product line platform can be a risk.

Possible indicators for this risk are:

- The components shared among other applications are actively used while not fully matured

3.4.11 Risk of unclear requirements

When requirements are unclear it may not be upfront clear what modifications are required on the platform in the near future. This may happen, for example if later in the development cycle the intentions of certain requirements become apparent. Or when it becomes apparent that some requirement has more impact on a product line platform after a client has signed a fixed-price contract. This impact may for example include unexpected changes in other products depending on the platform.

Possible indicators for this risk are:

- Separation of responsibilities in the requirement engineering activity
- Large products with large number of requirements
- Long iterations/large release scope

3.4.12 Risk of different interpretations of artifacts

When design artifacts like requirements, planning documents, or documentation of components have to be used by other project teams there is a chance that elements from the artifacts are misinterpreted. This may cause deviations in implementations, but also to differing expectations of involved parties.

Possible indicators for this risk are:

- Distributed work over separate teams
- Design documents are shared
- Large development scope

3.4.13 Risk of goal ambiguity

Let's assume a developer has build a component in a platform with a certain goal or purpose in mind. Another developer wants to add a feature to this component for his project, which conflicts or diverges from the goal the original developer had in mind. But, the other developer does not know about the goals of the original developer and starts implementing the feature according to his own ideas.

Possible indicators for this risk are:

- Large scope of work, large number of development teams
- Little documentation or communication of component's design goals
- Little effort in keeping design documents up-to-date

3.4.14 Risk of scope widening

At a certain moment a customer is brought in for whom a new product is being developed on an already existing platform. Because the business wants to offer this product to more customers in the future, the platform is expanded with the consequence that new components must be build and existing components require modifications.

Possible indicators for this risk are:

- Platform with mainly immature components
- High chance of new prospects with unique desires or from a different domain
- Platform is not yet at a financial break-even point

3.4.15 Risk of scattered functionality

If components from a platform are not located in the same repository or when certain functionality is implemented scattered across different components, we speak of scattered functionality. Modifications on these components may potentially requite additional modifications in multiple locations.

Possible indicators for this risk are:

- Platform components are spread across multiple repositories
- Core platform functionality is implemented across multiple components

3.4.16 Risk of delocalized plans/documents

When a platform road-map or planning is maintained at different locations, agreements have to be made regarding how these documents are managed. For example, let's assume the design of an artifact of a new feature is scattered across multiple locations. How do you coordinate this?

Possible indicators for this risk are:

- Separation of work of system design across scattered teams
- Large scope of work
- No or poorly incorporated central storage of design documents

3.4.17 Risk of iteratively changing reuse components

If a component is build in multiple iterations, for example because the requirements of a platform are not yet completely clear. These components can be exposed to a high level of change, resulting in an unstable platform. This is an undesired situation if the product is already live.

Possible indicators for this risk are:

- Agile development methodology employed
- Fast time-to-market, business value thinking
- Using immature components in production

3.4.18 Risk of changes in product line assets at the product level

Functionality required for a specific product which will be implemented in components of a platform or for which platform components have to be modified can have undesired affects on other products of the platform.

Possible indicators for this risk are:

- High level of coupling between product-specific assets and platform assets

3.4.19 Risk of enhancement to a cross-cutting concern

The more overlapping functionality a shared component has with other components, the more impact a modification on this component can have on the platform and its depending products.

Possible indicators for this risk are:

- Low level of modularity of components
- High level of coupling between platform components and product-specific components
- High number of hubs in the dependency graph of the platform

3.4.20 Risk of component granularity

Large components can be very incoherent, which makes them less suitable for reuse in a platform. The bigger the components, the larger the chance that they may be affected as the result of a modification.

Possible indicators for this risk are:

- Large components with low level of cohesion and low level of orthogonality
- Configuration of functional modules affect multiple components
- Functional modules are implemented as a collection of components

3.4.21 Risk of circular dependencies

Let's assume component A utilizes functionality of component B and vice versa. The more of these 'circular dependencies', the less modular the components become. Less modularity render components less suitable for re(use) in a product line platform. A high level of circular dependencies can make modifications or testing of components very difficult.

Possible indicators for this risk are:

- High level of circular dependencies in core platform components
- High level of circular dependencies between platform components and product-specific components

3.4.22 Risk of non-standardized configuration interfaces

In the case of multiple products having multiple build or configuration interfaces, fast testing of changes across multiple products may be hampered if there is no automated option to do this. If two products use a different build-system (for example Imake and Automake) and executing an install-command invokes different behavior on both systems, this can result in problems.

Possible indicators for this risk are:

- Different deployment or configuration technologies used for platform products

3.4.23 Risk of early binding of build-level dependencies

If external components are using hard-coded relative paths, it delimits the boundaries of usage of the component and at what level of detail or granularity the location of the components are defined. The more generic this is designed, the more flexible component can be utilized and the harder modifications of dependencies are to work.

Possible indicators for this risk are:

- Hard-coded configuration on product-level of dependencies
- Dependencies are managed by hand

3.4.24 Risk of making a composition by hand

Releasing a product of a product line by hand can be very time-consuming if a lot of folders and files have to be bundled together across multiple locations. The more complex this task, the higher the chance errors are made when for example some file has been forgotten to transfer.

Possible indicators for this risk are:

- No automated compile and deployment technology used to compose a platform product

3.4.25 Risk of making an application/component reusable

Sometimes a product-specific component can contain functionality useful for other products. To enable this component to be offered to other products, it has to be made available as a generic employable component. Most of the time this requires refactoring of the component, with all possible negative effects.

Possible indicators for this risk are:

- Large number of outdated classes or components which actually need a good refactoring job for recent demands
- Tight coupling of classes or components

3.4.26 Risk of heterogeneous communication

If there is little to none communication or process defined as to how to deal with modifications of platform components, all sorts of undesired things happen which may impact other products without any checks or taking of responsibility.

Possible indicators for this risk are:

- No communication policy when updating platform code
- No defined process or definition of responsibilities regarding modifying platform code

3.4.27 Risk of centralization in group based collaboration networks

If knowledge questions regarding a platform always go via the same person it creates a bottleneck in the process. The dispatchment of changes or problems depends on the information availability, the efficiency and the skills of a single person.

Possible indicators for this risk are:

- A central person of authority sitting on the shared code base
- A development culture where the central person is given the responsibility of designing and maintaining the shared codebase

3.4.28 Risk of reuse experience level

When developers have a high level of experience with developing platform components, they can more easily oversee the impact of modifications. Also, there is less chance of introducing errors in other products. More experience may also reduce the risk of having less modular components, hence more flexible components.

Possible indicators for this risk are:

- No knowledge sharing of platform components
- Junior developers with high level of design responsibilities towards platform components

§ 3.5 CONCLUSION

In this chapter we answer the first research question of this study: *what are the risks of a shared codebase environment with respect to changing requirements?* We conducted a literature study of industrial case studies where we found 28 risks, each discussed with their possible indicators.

- Chapter 4 -

Interviews

§ 4.1 INTRODUCTION

The second main research question of this project is: *What approaches or techniques can be used to mitigate risks in an agile shared codebase environment?*. In section 2.4.1 we stated that this will be done by interviewing employees at Topicus in different roles from different projects at different business units.

To answer this question, 8 interviews have been conducted among 3 different business units of Topicus. In this chapter the goals of the interviews are stated, the design of the interview questions is discussed (section 4.3) and the used interview protocol is described (section 4.4). The results of the interviews can be found starting from section 4.5.

§ 4.2 GOALS

From the interviewees, we want to know the following:

1. What issues with regard to a shared codebase and changing requirements do people encounter?
2. What is the daily practice of working with a shared codebase?
3. What have been the experiences with the daily practices?
4. What would people see differently in this regard?

§ 4.3 DESIGNING THE INTERVIEW

Before formulating the interview questions, in this section some theoretical background is given to justify the choice of conducting interviews. This is done by discussing what types of interviews we can choose of and how data then can be interpreted.

4.3.1 Why interviews?

For this project we want to assess common issues and approaches across different business units based on experiences from the past. But we only partly know what kind of information we are looking for. Also, we are not trying to prove some hypothesis and we are not interested in a comparison of demographics. Taken together we think that a qualitative study design is the most appropriate study design.

We can distinguish between the following qualitative study designs: case studies, comparative studies, retrospective studies, snapshots, and longitudinal studies (Flick, 2009). Snapshots and longitudinal studies are more focused on analyzing the effects behind some phenomenon, which is not the topic of this study. Also, we are not so much interested in what business unit is better than the other, but more what works well under what conditions. So a comparative study would not suit this study well. Rather, the study should be designed as a combination between a case study and a retrospective study. We want to analyze experiences from the past which we will do with interviews among employees at Topicus. While we also could have chosen for questionnaires, we agree with Lindlof et al. (Lindlof and Taylor, 2002) that “[interview’s] ability to travel deeply and broadly into subjective realities” is something required for this study, since we are interested in the best practices from inside Topicus based on the experiences from employees.

4.3.2 Interview type

There are a number of different types of interviews we can use for this study. Flick (Flick, 2009) states the following types: focused interview, semi-structured interview, expert interview, problem-centered interview, ethnographic interview, narrative interviews and group interviews. In this study we want to look for best practices which are stored in the actions and thoughts of the employees. To get this information on paper, we need up-front knowledge of what kind of information we can expect so directing questions can be asked. We are not interested in individual episodically data, but situational experiences. However, these experiences are subjective and thus still personal and we can expect that not everyone is willing to share these. Directing follow-up questions as well as confronting questions for this purpose can a good tool to drill deeper into the situation, but we must not focus on the individual too much to keep the focus on the processes. So in summary, we need an approach which enables open questions while still being able to focus on specific categories of information. Also we need to get into the thought process of the individual, without focusing too much on the socially or biographically aspects. Using Flick (Flick, 2009) we then conclude that a semi-structured interview approach seems the way to go here.

A semi-structured interview is built around hypothesis directed open questions or on theory based directed open questions. Usually this type of interviews is used in the reconstruction of subjective theories. Problem with this approach is that interpreting the data can be difficult since the resulting verbal data is mostly uncategorized and requires a lot of effort to transcribe and analyze. It is a trade-off between getting significant comparable data (hence, closed questions) or allowing for asking follow-up question based on answers of the interviewee.

4.3.3 Interpreting data

Interpreting transcriptions of (not only semi-structured) interviews can be done in different ways, but mostly is done by coding. One of the most influential models for coding qualitative data is the grounded theory approach (Lindlof and Taylor, 2002). Flick (Flick, 2009) discusses a number of coding approaches like open coding, axial coding, selective coding and thematic coding. For this study thematic coding or selective coding seem best, but the effort this will cost seems a bit too much for the goals of this study. We will apply a pragmatic coding technique where using 5 colors the transcriptions are analyzed. In our transcriptions sentences can be marked *Red* to indicate a problem, *Green* to indicate a chosen solution, *Blue* to indicate a wish or desire, *Purple* to indicate an important statement and *Orange* to indicate important contextual information. Then, in the side-line of the transcriptions near the marking the highlighted quote is summarized in a few words.

§ 4.4 INTERVIEW PROTOCOL

We conducted, audio recorded and transcribed 8 interviews among 3 different business units. All interviews were conducted on-site and were semi-structured. The selection of candidates was done via a snowball approach where via-via people were contacted and recommended based on the desire of wanting to speak team-leaders, developers and analysts with experience in working in shared codebase environment. All interviews lasted around 1 hour. A number of standard questions were defined which at least should be asked and a listing of themes was created before the first interview which was used to ask follow-up questions and starting point for new questions. The design of this listing of themes is discussed in the next section. The used protocol can be found in Appendix A.

Also, 1 pilot interview was done at FinCare to test the questions for completeness and to practice conducting such interviews.

4.4.1 Shared codebases challenges and requirements change

The goal of this section is to determine what characteristics of requirements changes are relevant in a shared codebase environment in terms of assessing the associated issues. We do this by looking at the stated challenges from Ghanam et al. from section 2.3.2 and how they relate to all the characteristics of change discussed in section 2.2.4.2.

4.4.2 Heuristic

The result is a table of characteristics linked to a number of challenges. We will use this as the input for designing interview questions for this project. We used the following heuristics for this assessment:

A challenge is related to a characteristic if

- the challenge' severity is different for different values for a characteristic
- or the effectiveness of dealing with the challenge depends on the value of the characteristic
- or value of the characteristic depends on the challenge (its severity or when dealing with it)

Applying this heuristic yields the result found in Appendix A. The top selection of both dimensions of the table sorted by the number of matches based on the heuristic:

Sorted challenges:

- | | |
|----------------------------|----------------------------------|
| 1. Reuse | 8. Instability |
| 2. Continuous integration | 9. Product ownership thinking |
| 3. Release synchronization | 10. Standardization of documents |
| 4. Testing | 11. Requirement of combination |
| 5. Cross-cutting concerns | 12. Platform quality |
| 6. Decision-making | 13. Platform stability |
| 7. Business-value thinking | 14. Business strategy |

Sorted characteristics:

- | | |
|----------------------|---------------------------|
| 1. Criticality | 6. Domain |
| 2. Project phase | 7. Developer experience |
| 3. Manager's control | 8. Motivation/opportunity |
| 4. Granular effect | 9. Time of change |
| 5. Trigger/source | 10. Frequency |

4.4.3 Interview questions

Based on the sorted challenges a semi-structured interview was designed. The used protocol is included in Appendix B. In Table 4.1 a mapping is given of the different topics and the concrete interview direction used in the interview sessions.

Topic	Interview directions
Background interviewee	Ask about current position and responsibilities
Change handling activities	Ask what kind of changes come across, how does the process look like
Change representation	Ask about documentation of changes
Reuse	Try to get a picture of the platform layout, goal of the platform and components
Continuous integration	Try to get a picture of the general development approach
Release synchronization	Ask how the release cycle looks like
Testing	Ask about the chosen testing techniques and what is tested
Cross-cutting concerns	Get a picture of the relations/dependencies among the different products
Decision-making	Ask who makes or can make platform related decisions
Instability	Ask if and how often there is platform instability
Product ownership thinking	No explicit question
Standardization of documents	No explicit question
Requirement of combination	No explicit question
Platform quality	No explicit question
Platform stability	No explicit question
Business strategy	Ask what the expected road-map for the products is
Requirement of maximum reuse	No explicit question
Accessibility	Get a picture of the knowledge sharing activities of components

Table 4.1: *Relation topics and interview directions*

§ 4.5 BTOPP-MODEL

In this section we define the structure used to discuss the results from the interviews. The basic assumption here is that interviewees mention aspects related to a particular organizational perspective. Garcia et al. (Garcia et al., 2007) define 4 perspectives for reuse in the context of defining a maturity model for reuse adoption. These perspectives are: *organizational*, *business*, *technological* and *process*.

According to Garcia et al. the *organizational* perspective includes all “activities directly related to management decisions to setup and manage a reuse project” (Garcia et al., 2007). The *business* perspective includes all “issues related to the business domain and market decisions for the organization” (Garcia et al., 2007). The *technology* perspective includes all “development activities in the software reuse engineering discipline and factors related to the infrastructure and technological environment” (Garcia et al., 2007). And finally the *process* perspective includes all “activities that support the implementation of the engineering and the project management practices.” (Garcia et al., 2007).

These categories were also used by Lucredio et al. (Lucredio et al., 2008) in the context of a case study identifying key factors in adopting a reuse program. the categories were not strictly defined, but used to structure the aspects found from the study. *Organizational factors*: software organizations and team size, project team experience, software reuse education, rewards and incentives, independent reusable asset development team. *Business factors*: product family approach, kind of software developed, application domain. *Technological factors*: software development approach, programming language, repository systems usage. *Process factors*: quality models usage, systematic reuse process, kind if reused assets, origin of the reused assets, previous development of reusable assets, specific function in the software reuse process, software reuse measurement, software certification process, configuration management of reusable assets.

In 1991 a model by Scott-Morton was introduced to model the relation between organization (structure, culture), business strategy, technology (employed IT landscape) people (skills, experience) and the business processes (management practices, procedures) (Scott-Morton, 1991). The model is depicted in Figure 4.1.

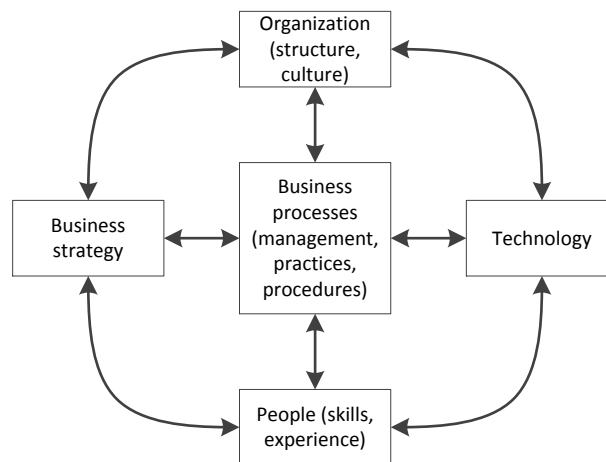


Figure 4.1: *BTOPP model*

To structure the results of the interviews we use the categories of Scott-Morton, but we give them slightly different meanings based on the interpretation of Garcia et al. and Lucredio et al. by putting the categories in a more software product line development perspective.

- **Business aspects:** activities and issues related to the business domain and market of the software product lines
- **Organizational aspects:** activities and issues related to setting up and managing the shared codebase of a software product line.
- **Process aspects:** activities and issues related to development and maintenance of the shared codebase and thus indirectly the individual software products.

- **People aspects:** activities and issues related to the skills and experiences of the people involved in the development of the software product line.
- **Technology aspects:** activities and issues related to the technical environment required for supporting the development and maintenance of a software product line.

§ 4.6 THE INTERVIEWED BUSINESS UNITS

The 8 interviews have been transcribed and coded according to *problems, solutions, contextual remarks, wishes* and *generalized statements*. As said before, to structure the results, the coded elements have been structured according to the BTOPP-model as discussed in section 4.5. We tried to find themes or aspects raised by the interviewees and discuss them using statements from the interviews. In the interviews we distinguish between the business units A, B and C. Their cases and situations are briefly discussed below.

4.6.1 Unit A

At unit A one team leader and one senior developer have been interviewed. The components at unit A can be described according to 3 levels. The first level really is a shared component base for everyone. It is called ‘*COBRA*’ and it holds around 1000 generic components with a total of 110.000 lines of code. These components can be further structured in a number of modules. These modules include functionality like: forms, data-panels, bootstraps, web-services, database handling, pdf-conversion, reporting services, security, database migration, web-components for the front-end and a number common functionality like for example logging. Besides COBRA there are a number of separate components for handling interfaces with external (mostly government) systems. The third level is codebase shared among two of their products from the same domain. This codebase consists of common functionality of the products which is not shared outside of the scope of the two products. The structure as described above is depicted in Figure 4.2.

4.6.2 Unit B

For unit B one lead architect, two team leaders, one developer and one analyst have been interviewed, spanning 3 different projects. The codebase does not have a component repository as unit A has, yet a distinction can be made between generic modules, domain specific modules and generic components.

4.6.3 Unit C

At unit C one analyst has been interviewed. Unit C, in contrast with unit B and A, is more involved in one-off projects than SaaS projects like the other two units. Still their codebase consists of components which are reused (or redeployed) in their projects, but not shared like unit A or utilized to have a SaaS platform like unit B.

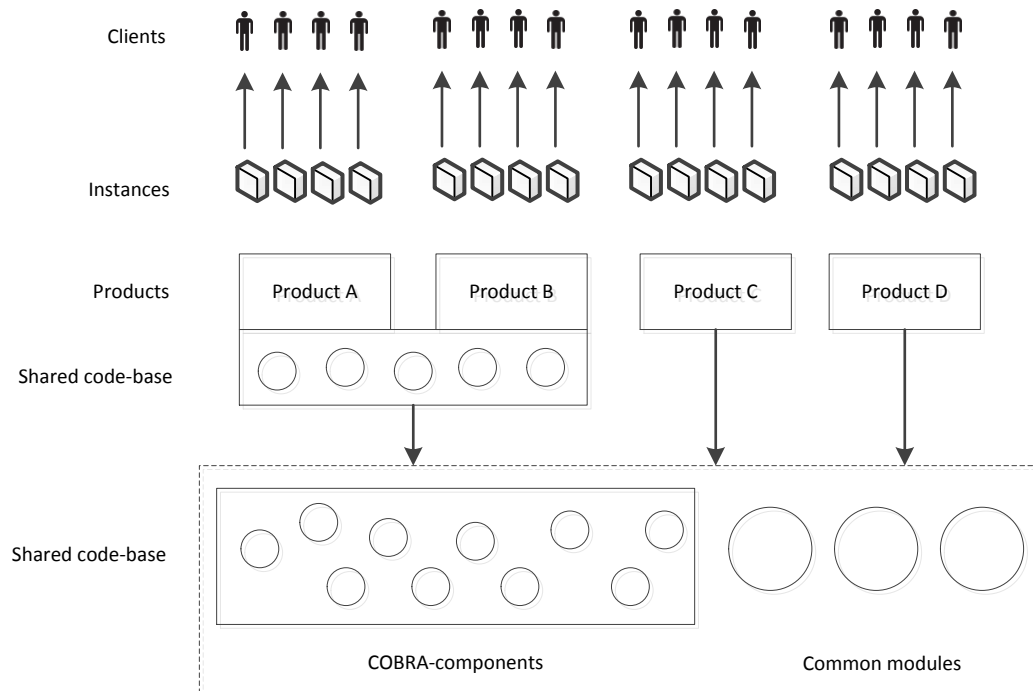


Figure 4.2: *Codebase unit A*

§ 4.7 INTERVIEW RESULTS

In this section we present the found aspects from the interviews across the three business units. The aspects have been structured according to the BTOPP model from section 4.5: business aspects, organization aspects, process aspects, people aspects and technology aspects.

4.7.1 Unit A

4.7.1.1 Business

Different domains Having multiple products in different domains can cause a clash about stakes. In unit A customers from different sub-domains are served with the same kind of application. Still, they have separated the projects and work on a separate codebase to allow separate domain concerns to be implemented and allow the products to exist in different life-cycles.

Who pays? When multiple projects are using and developing on a shared codebase, someone has to pay the bill. In unit A development on a shared component is paid from the budget of the project doing the development. An interviewee of unit A about this:

“Sometimes you just start shifting hours. As long as there are multiple persons who work on the components, it does not matter that much. Every project has someone who edits the shared stuff. [...] but, sometimes yes, the hour distribution among projects is not fair.”

Another interviewee at unit A about this topic:

“Product A has the most up-to-date component library. This is logical, since this project uses the most and is under active development. We often have the discussion

that additions to components are from a desire from project A, but these additions can also be beneficial for other projects. We are getting money for this from our customers, so the resources come solemnly from project A. This is sometimes very unfair. [...] It keeps important refactoring activities and updates from happening.”

Legal aspects When sharing components among different business units, it must be made clear who legally is the owner of the component. For Topicus this is especially important since all units are registered as separate companies under the holding Topicus, hence created assets are not automatically legally owned by everyone in the holding.

4.7.1.2 Organization

Cross-cutting concerns A downside of sharing code among multiple projects is that editing something on a shared component can break something in another project. According to one interviewee at unit A this can be a risk since the impact of a change is not always known and you always have to deal with desires from multiple angles. On this topic one interviewee of unit A says:

“You just can’t make major changes [when working with shared components]. If I do a commit on COBRA and [product A] is having a release tomorrow and thus have made their own branch of COBRA, I can easily have broken something on [product B] with that commit. But since [product A] now works on a different branch, the bug can be introduced unnoticed in [product B].”

But he also says:

“[...] in the end we all are very pragmatic and fortunately smart enough not to make compromises which may cause problems in the future. In practice, we do not encounter that much problems in that regard. [...] A design is made from the team where the change originates from and if there is even the slightest hint that something may be impacted you check the impact.”

Responsible person In unit A there is a central person who took it upon himself to be the responsible person for the shared codebase. In unit A there is an active discussion about if this requires versioning of the codebase, having release cycles for the shared codebase, testing of releases and back-patching.

Outsourced service desk In unit A the service-desk and implementation of the products at customers is out-tasked to a third-party. This is done deliberately to have a more strict separation of responsibilities. The number of clients for unit A is rather large, which makes having a number of levels of responsibilities between end-user and development teams a necessity. The third-party handles incoming bug reports and judges them as relevant or critical or not and if required passes the report on to unit A. There a dedicated person can pass a report on to a project teams as a bug or a feature request. An interviewee of unit A on this topic:

“As a software development organization, we like it not having to deal with everyday end-user problems of customers. With all due respect, most of the time the issues they raise are really not that relevant. However, we notice that having a helpdesk as a buffer is more handy if you can actually walk to them for for example clarification about an issue. The helpdesks of our products are located in different cities, making that impossible.”

The downside of the outsourced service desk is that sometimes when a bug is small and easy to fix, the layers of communication can cause misunderstandings and a long time-to-fix. One interviewee indicated that having a support-desk on the same floor is so easy, since within a few steps a developer is reached and the problem can be solved.

Gatekeeper One interviewee, who is a team leader at unit A, indicated that he as a team leader is the gatekeeper for updates. When a developer wants to update a library for a project, the team leader enforces the approach that the developer has to convince the team leader of the necessity of the update. Also, the dedicated person in unit A is the central person who people confer with before pushing important updates, thus serving as a sentinel for the shared codebase. An interviewee about him:

“He marked himself as the owner of the codebase. [...] He sees the most, if not all, updates on the components. This means that you directly have some level of quality check on the shared codebase.”

4.7.1.3 Process

Release-cycle Most products at unit A have a release-cycle of 4 releases per year. Four weeks prior to a release the product is put on an acceptance server. The implementation partner then is responsible for testing the release. If the release is accepted, the release is placed on the production server for the customer.

Also, every day a snapshot of the code is placed on a test server. The test server is normally only used internally for the software developers, but when a product is immature and under active development, the customer can have access to the test server for fast feedback and demonstrations of the latest developments. When the first stable production release has been done, customers usually no longer have access to the test servers. To still allow fast customer feedback and demonstrations of the latest features, unit A has employed a simulation environment for their products. The reason for not allowing customers anymore on their test servers is also that after the first stable production release the customer base is expected to grow. Also, acquisition and implementation of customers then becomes the responsibility of the implementation party. Allowing customers who were not involved in the initial release to be closely involved in the development process then does not make sense. Sometimes this still is done, but it all depends on Service License Agreement (SLA) the parties signed. One interviewee on this topic:

“In general we can give everyone access to [the test environment]. When customers are running the production release, they don’t have their own copy of their data on the test server. So, if we want them to test something we only can do that by giving them access to our [simulation environment].”

The downside of the 4 releases per year is that when issues are found on the acceptance server during testing, patches have to be done directly on the acceptance server. The development branch is already 2 months ahead of the acceptance version, so the codebase is frozen. The same interviewee at unit A about this:

“After the release on the acceptance server is the moment the customer can actually start testing the product. This also means that findings have to be patched directly on the acceptance environment. The code on this environment is frozen, so changes can only be done regulated. For couplings with other systems this sometimes is difficult. From that moment you just cannot make large changes. Also, this is the moment they start testing with their own data. For customers in the

production-phase who want custom couplings with their systems this approach can be problematic.”

Life-cycle of products In unit A they have the case that two products have the same codebase and also use COBRA-components. One of the two products, product A is relative mature and stable. the other product, product B, is very young, serves a different domain but is for the most part based on the same code. The difference according to one interviewee at unit A:

“When you are in the same life-cycle with all customers from a domain, you have to maintain different dependencies. From the perspective of [product A], you can’t just start patching. There has to be a RFC (Request for Change) which must be approved. Then only a patch on the production server may be done. For [product B] still a lot of customers are just going live, it is a fairly immature product so more patches are required. But since both products use the same codebase, this requires coordination among the two teams.”

Different domains When products depend on the same codebase, but are operational in different domains, terminology difference between the domains may cause issues. In unit A issues like naming table attributes or generic labels for forms were mentioned.

Internal collaboration At unit A the teams of product A and B meet once every 4 weeks to discuss spanning issues between the two projects. For small issues having separate teams coming together once a month may be insufficient, since there is a mental barrier to just walk over to the other side to ask someone personally. It only works when in-between two meetings you can have regular ‘coffee’-moments to discuss small issues while waiting at the coffee machine. This is one of the reasons they have an issue with an external implementation partner who is physically located somewhere else. To mitigate this to some extent, unit A often has an database expert from the implementation partner on-site.

Knowledge about components At unit A it was also said that there is not much knowledge sharing about components. Components are created and used from the perspective of a single project and forgotten after a while. One interviewee on this topic:

“Your own project uses [the component], but it is not like people are actively promoting that they have created a new component. Not like: hey listen, we have a new component here, go use it. It has to fit of course, but knowing about the existence of certain components, that could be improved.”

The interviewee also has a suggestion as to how to achieve this:

“I want more knowledge sharing, not only of components, but also of applications. [...] you should have demonstration session or knowledge sharing session, spanning all projects. Showing, last month we did this and that, this is what it can do. If you’d like to use it, you can find it in COBRA.”

Testing At unit A testing is done using unit tests, regression tests and Selenium tests. Both interviewees said that they regarded the coverage of the unit tests very low. Selenium tests are usually meant for front-end testing of interfaces to see if for example by clicking thus button if some information is displayed on the screen at the correct place. unit A uses Selenium differently,

they use it to automatically access all pages and follow all links and just see if no stack-traces occur. The unit tests are seldomly made and if they are made, then only for a specific project. An interviewee on test-cases:

“We have test cases in the projects, but rather only if the developer’s felt like to do so, then maybe he wrote one.”

There are no corporate policies or standards as to what to unit test and what not. Some core components have unit tests however, but not much. The selenium tests and unit tests are run automatically when a daily snapshot release is build. Some functionality is tested using regression tests, which basically means that a number of scenario’s are executed when releasing a new version of the product.

Communication of changes An issue raised at unit A is that changes sometimes are not communicated well. A specific problem here is that an issue on a shared component may already be fixed or being fixed at one project, but another project is also starting on a fix. Or sometimes it already may be solved by a team without communication this.

4.7.1.4 People

Pigheaded developers Developers can be pigheaded about solutions instead of pragmatic. One interviewee about this on the context of sharing experiences with developers in knowledge sharing sessions:

“[...] the discussion sometimes can linger on, we just have a couple of pig-headed people here who think like: well, why don’t you just do this here. And they then cannot see that such a solution may not be pragmatic.”

4.7.1.5 Technology

Interfaces and web-services At unit A a lot of custom interfaces are maintained with systems from the IT landscape of customers. An interviewee at unit A about this:

“Every [customer], because they are rather large, they want their own custom-made additions. these additions are often related with integration of the product with their IT landscape. This often means a lot of small projects to implement small interfaces with their local systems.”

Another interviewee at unit A about this:

“For product A we look at the client’s systems and create and maintain a lot of interfaces with them. I do not favor this, what happens often is that one specific web-service is build for one system of one school. Currently we have 30 customers. And eventually I expect we will have around 50 customers, so with every school having multiple systems you can calculate the number of interfaces. They all look the same, but differ always on some points. So what happens when a new interface is build, a previous similar looking one is copy-pasted and edited.”

For a new product they want to do things differently as the interviewee explains:

“For the new projects we have said, we develop one set of web-services and that is all [customers] have at their disposal. Often they just want to extract information from the system, so if we design our web-services such that everything can be extracted, no customer-made interfaces are required.”

Libraries Large common libraries have high potential to break down projects. These libraries are often external libraries who are not maintained in-house, but still can have regular new versions releases. At unit A when a new version of such a library is released, the central component contact usually initiates such updates. He then looks around at other projects to see what the impact of the update will be and he just makes sure that the transition to the new version goes well. It can happen that the impact beforehand is underestimated and that small updates may escalate to large issues. There is no formal approach used here, the developers rely on experience and skills. A wish for a new version of some library may also come from a project it-self, but in all cases a convincing case has to be made as to what the new update will bring to new all products. The number of fixes and new features in a new version may be marginal, but often it is wise to stay up-to-date. Old, outdated libraries on which still a lot of products depend are much more difficult to migrate to a new version than updating a more up-to-date library. This for example happens at unit A when an old library is used in a production release and a new version is used in the development environment. An interviewee on this topic:

“Updating a common library is a lot of work and making a selection of updates for a patch is really hard. [...] when you get a [library X] update, you also have to update [library Y]. [...] so, that is one of the down sides of this approach. I’d rather see that we release more often, once per month, maybe more. But we can’t, because of our SLA.”

Databases At unit A having different database technologies in production may require the same query to be written twice in a different notation. The reason for having different database technologies in place is a legacy issue, but something they still have to deal with. This sometimes makes it difficult to create components, since different query notations need to be supported.

Access rights One of the issues of shared functionality is that you don’t want to specify customer-specific access rights in the shared components them selves. At unit A they therefore generalized the access rights of functionality and separated it from the components.

4.7.2 Summary unit A

4.7.2.1 Issues

In Table 4.2 we abstracted the issues from the interviews and related them to the risks from section 3.4.

BTOPP-element	Issue	Associated risks
Business	Different domains	1, 5
	Who pays?	-
	Legal aspects	-
Organization	Cross-cutting concerns	15, 19
	Responsible person	27
	Outsourced service desk	2, 5
	Gatekeeper	27
Process	Release-cycle	2, 5, 18
	Life-cycle of products	10
	Different domains	19
	Internal collaboration	16, 26, 27
	Knowledge about components	16, 12
	Testing	18, 19

BTOPP-element	Issue	Associated risks
	Communication of changes	26
People	Pigheaded developers	-
Technology	Interfaces and web-services	4, 19
	Libraries	19
	Databases	19
	Access rights	19

Table 4.2: *Interview results: issues*

4.7.2.2 Solutions or best practices

In Table 4.3 solutions or best practices are listed abstracted from the interviews.

BTOPP-element	Solution/best practice	Associated risks
Business	-	-
Organization	- Dedicated person for shared components.	26
	- Other units can have read-access on codebase, but no commit rights.	13, 15, 17, 18, 19
	- Two-layered bug-reporting structure.	2
	- Third-party implementation partner.	2, 26
	- Testing across products responsibility of implementation partner.	-
	- Separation of development, support and implementation in different organizational parties enforces separate responsibilities.	26
Process	- Getting know-about of of new components or features by looking over the shoulders of people outside your project.	13
	- End-user meetings for collective feedback.	-
	- Make branch of shared codebase before a release.	18
	- Prevent from doing small patches on production.	17, 18
	- Prevent releasing new features or sub-sets of features before a new 'big' release.	-
	- In early development stage give customer access to a test environment for fast feedback.	-
	- Later in development stage give customer access on simulation environment.	-
	- Everyone can edit/add new COBRA-components.	27
	- Expert on-site of implementation partner for fast feedback of technical details.	-
	- Scheduled monthly meeting with developers cross-projects.	13
	People	- COBRA is 'of, by and for everyone', creating social self-regulating system.
Technology	- Selenium to test front-end of products.	10, 18, 19
	- Unit tests for testing of core-web-services.	10, 18, 19

BTOPP-element	Solution/best practice	Associated risks
	<ul style="list-style-type: none"> - COBRA with generic components for everyone to use. - Automatic nightly snapshot builds with automated running and reporting of unit tests and selenium tests. 	- 10, 18, 19

Table 4.3: *Interview results: solutions or best practices*

4.7.2.3 Desires

BTOPP-element	Desire	Associated risks
Business	A clear policy on how to bill work on shared code	25
Organization	When working with other organizational units, working in the same physical location is preferred	10, 11, 13, 15, 19
Process	<ul style="list-style-type: none"> - More awareness that when large modifications are done, this can have an impact besides your own project scope - More knowledge sharing sessions about both components and applications - More communication when someone is changing a shared component 	26 13, 17, 25 26
People	-	
Technology	More information of coming updates on external components	19

Table 4.4: *Interview results: desires*

4.7.3 Unit B: issues

4.7.3.1 Business

Different domains Also in unit B there is a difference in the domains and how things work. Unit B delivers two (but not only) products in the health care domain, one for health care chains and one GP offices. For the latter a break down is much critical than for the first. One interviewee about this:

“The GP office software is for emergency situations. The health care chain software is, well, if it breaks down you make a note and call the help desk the next day. [...] These customers are less faster angry. Which can make you a little slack sometimes.”

Volatile market When operating in a volatile market, making extensive test cases and investing in components is very difficult to justify financially. One interviewee explains:

“You often have the feeling that we can spent 3 months to make everything perfect but if then only one customer uses the function then a total refactor may be required because some new laws or regulations.”

4.7.3.2 Organization

Management pressure A down-side of shared components in the same domain is that the effort to utilize some component in a product can be underestimated. One interviewee of unit B on this:

“For example, our director comes from another unit and says: I have seen they do this and that, we can have roughly the same by tomorrow. [...] Yes, you can if you have all the component around it also and use them the same way as they do, which we don’t. ”

He therefore argues about more a focus on modular units of functionality in such a way that these units really can be moved around. The reason for this is also commercial:

“If we would focus more on modules, more on cohesion of components which also have a front-end implementation than [utilizing] them would be a lot easier. Components [instead of these modules] are really born from a technical perspective. Developers think in components, customers think in modules.”

4.7.3.3 Responsible person

One of the issues raised is that it is difficult to say exactly who is responsible for what component. One interviewee on this:

“I can’t pinpoint who’s responsible for what component, this requires close knowledge of what is happening to the components.”

This indicates that knowing who to go to for some issue regarding a component starts with knowing what components are under edit and from what context. One of the solutions could be to assign one person to be the gatekeeper of the codebase, but this might not be comprehensible as the same interviewee indicates:

“We have too many components to really have a single person responsible for all components.”

Another interviewee on this:

“[One responsible person] may give you a lot of overhead and planning hassle, for example if the responsible person of some component is very busy with his own project, so then he should schedule some time for sometimes very some estimations.”

Another interviewee on component-based developing:

“Within a project-based structure, working dedicated on components is impossible. I can’t say to my boss, hey I’m going to spend 3 weeks on that component. [...] We sometimes have the discussion about making people responsible for some component, but often a deadline is approaching and such activities go to the background, because my project has to be finished so I don’t have time to make my components fancier.”

Justifying updates When in a situation like unit B which has a product with for example 7 propositions, putting updates on the table is hard to justify for customers. Refactoring activities required to enable the use of a newer library version which is not directly related to a new feature can be seen as either an ‘investment’ in the platform or ‘innovation’ or ‘maintenance’ activities. Topicus is a company who really wants to stay competitive, lean and transparent. Showing where money is spent on is very important. But Topicus also wants to be innovative, trying out new things, being bold. As one interviewee put it:

“The big question is, what are the benefits for the customer. Not so much probably. But I think [the update] is really cool.”

He also describes he in his role of lead architect tries to instruct the developers:

“What we try to learn our developers is: if the update is interesting then you should be able to explain to your product manager why. For example, because in the last period there were 10 bug reports on this topic. [...] I really have a heart for the technical side, but I need to be able to justify it.”

4.7.3.4 Process

Knowledge about components One interviewee indicated that knowing about component usage is not trivial:

“As a product owner, I don't have the knowledge of who is using what component. For the development teams this is something they should know. In general they know who is actively using components, because you see the commits coming along. But being 100 percent sure people are the only one using some component, no.”

Incorporating components If another team has some component available for reuse and you want to incorporate this component in your own project, you have to make an estimation regarding the expected amount of work and if the functionality you require can be met by the components. One interviewee on this:

“We expected that integrating the component was easy, now we have discovered that we still have to develop a lot on an application specific level.”

Refactoring There is a strong call-out for more refactoring, but the bottom-line is that in practice little refactoring is done. Refactoring can have the purpose of cleaning-up some part of the codebase, or to abstract some functionality and turn it into a generic component. The latter often causes the first to happen, but not the other way around. To have time and resources allocated for making a component tidy and generic, it has to have a direct purpose for some project. Outside of project's scope, the pressure is just too high to spend time on refactoring. Searching for opportunities for reuse is a spontaneous activity.

Justifying update Another aspect of justifying updates is that internally you must be able to explain why a component for something is required. For functionality regarding some standard this is much easier than for other kinds of functionality:

“The thing where there is a standard for everyone agrees on, that has to be put in a component right away.”

The kind of functionality where this accounts for mostly are standards for communication. Sometimes the justification for some modification is very legit, but still can cause problems:

“[Product A has an interface using [component X]. [Product B] also has [component X], but we needed just a little more interfaces some more abstraction, and that was not always communicated as well as it could have been.”

Cross-crossing concerns In unit B they work a lot with interfaces. This basically means that between users who use share some components the responsibility starts and ends by means of an interface, serving as a design contract.

“For some time we have been working with interfaces, so we have components which work with interfaces. This means that regardless of the person behind the interface, as long as it delivers this and that functionality, I don’t care about the implementation.”

But this approach also has a down-side:

“This means a lot of hassle, you always carefully have to think if you modify something, who uses this interface, what else may change. Sometimes we even discover that two different interfaces have the same implementation”

Something what makes finding out the impact of a change even more harder are bugs which are used as a features by other teams:

“What you also see very often is that someone communicates that a bug has been fixed and that another group calls out saying, hey, we made use of that, that ‘bug’ actually helped us.”

Of course this is a very specific situation, but it happen so now and then at unit B that new features affect shared components and other products. An interviewee about updating some components shared with another business unit:

“Problems with not communicating become bigger if you dont say in advance hey this and that is changed. [...] For the form component we had the situation that [Unit C] suddenly did an update. We used that components, they didnt know.”

Another interviewee on this:

“Very speaking example, I even dont know the exact situation if we mailed or they did. I think it was us, there was no more response and then suddenly something breaks down. Then you have an escalation, which happens sometimes. and you find yourself sitting with the project leaders, shit, how are we solving this.”

And yet another:

“When updating code from each other, sometimes the other is not aware of the update and the impact. So you need to communicate with each other.”

Testing Testing at unit B is done merely only on end-product functionality, so shared components don’t have dedicated tests.

“We rarely test on components. We always test the end-resulting functionality for our customers.”

Vision of components For a large set of components which can independently be put into action, it is difficult to maintain a coherent vision. One person at unit B about this:

“[...] you notice that it is difficult to keep 30 marbles in a wheelbarrow together.”

There is little vision of the components, where does it go to, why is it here. An interviewee on this topic:

“[...] that there is no thinking about, what is the general direction for this component, is it still relevant, is it logical to put this functionality in this component, shouldn't we have made two separate components from this components long ago?”

Another interviewee at unit B on this:

“Biggest problem is a difference in vision of what the goal is of some change. But also the impact on somebody else's project. Communication is vital here. ”

4.7.3.5 People

Interests In order to have innovation, in order to keep your platform fresh and evolving you need people who outside their job have an interest in the technology. In unit B most of the developer are, for Topicus, ‘old’, meaning that they are aged around 30. The result is that most developers are settling, starting a family and having less time and interest in spending additional time at home at their computer. One interviewee about this:

“We don't have the nerds, the gurus here, not many people who hobby at home and look for latest updates.”

Mental barrier There seem to be a mental barrier to communicate outside of the comfort zone of, not only developers, but also architects and analysts. In unit B, teams are sitting on opposite sides of the building, but in an open space with no doors. Still, staying at your desk and fixing it yourself is favored over walking to the other guy to ask something personally. Unit B works together with unit C on some components, hence sometimes communication is inevitable. One interviewee about this collaboration:

“Fixing it yourself is often done without communicating, because it is just a little faster. Even of the other guy is just 5 minutes walking”

Another interviewee on this:

“Problems internally can be easy solved by walking to the other side. But if you need code from each other, you can't just start editing it. There is the risk of doing it yourself.”

Domain knowledge When not knowing what issues customers encounter every day, how can make the right functional decisions?

“There is a lack of understanding about the domain. They have the freedom, but they lack confidence with respect to functional modifications.”

4.7.3.6 Technology

Component configuration For a SaaS delivery model flexible component configuration is important since it allows product-specific behavior without the need of implementing lots of product-specific components. However, the down-side of this is that it is hard for administrators to maintain, since the number of configuration options can explode. One interviewee on this topic:

“Configuration becomes more and more an issue to be able to flexible deploy components. [...] Which means you want flexible configuration. but if you have 5 parameters and an average project uses 20-25 shared components, it just is not manageable. Also, it makes it really hard to explain to the administrators. It really is a wish from a developers perspective, for administrators it can be a hell.”

Communication standards

“For communication standards, different interpretations of the standard are very maintenance sensible. Updates on the data happen very often and are, of course, never communicate, leaving us to discover such changes by ourselves.”

Splitting components In an environment where time-pressure is high, splitting components with the intention to merge them back later can easily grow into two very deviated components.

Email It was mentioned multiple time that email is a bad communication media for communication about changes on shared components. The main reason is that in email often a lot of information is hidden and you do not get instantaneously get feedback from your peers. Also, email is a very discriminating medium, since people are explicitly included and excluded. How do you know and decide who is interest in knowing about some update? And even if you would know, what are you communicating? Do you peers have any saying in the update?

“An email about an update is sent onto a mailing list for developers. It reaches everyone, but therefore it may be not specific enough for people to act.”

4.7.4 Unit B: solutions

4.7.4.1 Business

Transparency For unit B, transparency about license expenditure is important. All customers pay a license fee for the products, which means that when money is spent on new features, all parties must agree with the additions to some extend. Sometimes, a feature request comes from a particular client. For unit B it is then the task to put this request on the table and get the other customer enthusiastic for this feature. The vision of unit B is that the products will always stay service oriented and as little as possible custom additions for clients will be done. One interviewee on this:

“We always announce what we have on our release planning, so if they have some issues with it, they can give input. But of course, it has to fit in our vision. [...] It happens that a customer wants a custom interface with some system, fine you pay for it, but we also start looking if there are other clients who want the same functionality. [...] Strategically this is smart since we make clear that we don't just start spending license money on custom features.”

On the contents of the license another interview says:

“In our license we say that we have a financial part for maintenance work, for example when a new version of a framework comes out and we want to go along with this new functionality. [...] Also, changes in law and regulations can ripple through which may force us to do some additional work [...] From the license they pay us they have the right for 2 major releases each year with a few weeks of work. [...] For new features for one customer, we often just put it in our application, so that it becomes available for other clients as well.”

Future vision At unit B, a road-map for their products is maintained with the scope of 2 years. The road-map broadly states the vision of the products. The road-map is a lively asset, meaning that it is adapted to recent events and propositions, but more importantly, it is shared with customers.

Benefits

“My experience is that we until now, which has always been the direction we want to be in, we mainly benefit from the components. [...] We mainly see the benefits which are created because there has been a project which said Ill invest in this”

Another interviewee on this:

“I think the time investment required to make a component good reusable outweighs the benefits. Just fixing it only for your project seems faster now, saves time on the short run, but it does not in the long run. [...] Some domain generic component really won't change much over the years, like persondata. Customers just pay a certain amount of money and in return want a fixed set of functionality. They don't care if you put that in 10 components and merge it with your project or make it available for other applications. ”

4.7.4.2 Organization

Other units Unit B works closely works together with unit C on some components. The two units both have read and write access on parts of each other codebases.

Gatekeeper Every product-line has its own main-developer who decides what frameworks are going to be used. One of the interviewees described himself as a central authority on the main architecture of all applications:

“I'm a central person who actually sees both sides of the story and sees hey, we can reuse something here”

Cost sharing As mentioned before at unit A, it is difficult to determine who pays for the effort of working on shared components. Unit B regularly has strategic and operational meetings with customers to openly discuss the directions of platform. When some new feature pops up, it is put on the table before the customers. If there is enough support by all customers, the feature gets implemented and the costs are shared.

4.7.4.3 Process

Incorporating components How components are used to build applications is something which evolves over time. As one interviewee puts it:

“For [our new project] we have 20 balls, before we made 5 applications with 4 balls, now we make 1 application with 20 balls.”

But, as a number of interviewees point out, the negative side of reuse always overshadows the positive side. One interviewee on this topic:

“It is not always a nice fit. But often enough a fit which is usable and workable and, more importantly, something you don’t have to build yourself.”

And another interviewee:

“Sometimes reuse can be disappointing. When refactoring something, making it available for reuse, you must be able to abstract the functionality such that you are left with a piece of software which actually is of practical use. What happens is that it turns into something too abstract to use.”

The choice to start using components or to reuse existing pieces of technology can both be because of hard requirements or a design principle. However, in unit B there is not really a vision on how to use components. The initiative always has to come from the perspective and dedication of a project to make something available for other projects as well.

“If you only have to look from the perspective of your own project, the discussion becomes a lot easier. You don’t have any overhead, you don’t have to take into account the opinion of others.”

The discussion becomes easier when there is a clear mutual benefit for the teams to invest in a reusable component.

“Between teams or units interfaces with the outside world are good starting points for spotting reuse opportunities. ”

Developing for reuse, without a clear up-front benefit is also pointed out by yet another developer.

“When looking back, things are always decided with the knowledge of then. But the notion that in the end it is always financed from the perspective of a project. [...] Developing on a component is always from the perspective of a project. Component dedicated development often does not have a direct result. In general we don’t do things that do not have a direct result.”

Yet another interviewee on this:

“Afterwards making a component available for reuse has the downside that the original application still very much needs it.”

Emails We stated earlier that email can be a bad way of communicating about updates. However, they can also be very useful. some quote on email is used in unit B:

“Mostly emails are just send crisscross about who is working on what. Most easy way to discover who is using something is just to break it. ”

“Emails are only send for large updates”

“Emails are probing: where can I find framework X: ah on this and this branch.”

“When changing some component, we put it on the mailing-list.”

“Sending an email often is sufficient, sometimes you discover that months ago someone sent an email regarding an issue, but then an email has been sent and you can you refer to it.”

“Often such emails are not being send at all. Then some of your dependencies is being updated and suddenly your application is broken”

“Internal in Topicus warning someone about a pending update is not hard. But with unit C sometimes a mail conversation of over 10 mails can sprout, which is not really efficient.”

Development cycle While every team at unit B can follow its own course, there are some similarities between the teams with respect to the development process. First of all, when adding some new functionality a scope definition is made of 2 pages maximum in size. Based on this scope definition, which contains the basic outlines for some piece of work with the scope of approximately 2 weeks, sometimes a functional specification is made. One interviewee on this:

“Building from scope definitions instead of functional designs allows you to think outside the box of your project. A functional design is always product specific.”

Another interviewee on design documents:

“It is a bit the culture of Topicus, we dont do it the formal way. You dont have to deliver that technical design and get an approval form your boss, thats not how we do things. Maybe we went a little bit too far with that.”

In general, no technical designs are made. While some interviewees mention the added value of them, they regard the activity as too time-consuming and already outdated as the development starts. Also, in contrast with unit A, no snapshot releases are done of components. Unit B works with strict versioning of its components.

Every week they have a scheduled meeting with all developers of the unit. They spent an hour or something looking at results and messages of the code quality tool Sonar.

User feedback Unit B has regularly strategic and operational meetings with customers to discuss the planning and directions of the products. Release planning is shared with customers.

“If they have suggestions, they can say so, but it has to stay in the vision we have of the product. [...] Customer comes with feature request, we put it on front of the others.”

Also customers can submit a RFC (Request for Change) in a tool called Clienttell. Protopics has a service-desk for these kind of things. There a person judges if something is a bug or a feature.

At unit B they use a tool called UserEcho to get direct feedback from end-users. This tool is a small web-form in the application where end-users can submit suggestions and report problems. For these kind of issues they stimulate developers to spend 2 hours each day to pick up small items and fix them.

Responsibility When sharing code, the one working on the issue and updating some component should do the modifications and have the responsibility. Consumers of the component then can work on their own branch. Or, if required, consumers just make their own branch in the repository and work from there.

Other units Large components-libraries with common functionality are shared between two units B and unit C. Collaboration on components is done via interfaces. Implementation is their concern, as long as the interface contract is fulfilled. This requires actively seeking contact with people using and working on other components when modifications are done.

“You can’t test for modifications on a generic component for all applications. This will require you to test for all products on the platform. Without unit tests, you can’t just set up 5 different products and test it yourself.”

Release cycle Unit B currently follow a cycle of releasing a new version of an application every 1-2 months. They want to work towards a slightly longer release cycle of 4 releases per year, as unit A. The products are releases in 3 steps, each with their own lead time. Internally, the latest version is released on a test server in the form of daily snapshot. The next step is to release to an accept environment, where customers can have test the latest version. At a scheduled moment, the version of the accept environment is brought live in production.

Code quality time Every Friday at unit C, they have one hour scheduled to do some maintenance on their codebase. This hour is meant to be used as some quality time for the codebase.

4.7.4.4 People

Responsibility At unit B employees are relative more older, senior than the average of Topicus. According to one interviewee, this means you can rely on developers to take their own responsibilities:

“I expect people to take their responsibility when changing some component, without having a policy as how to deal with such an update. I trust them to do the correct things.”

Domain knowledge For developers to have some affinity with the domain and issues end-users experience, unit B is sending out employees to customers:

“I try to give employees affinity with the domain, but we are not there yet. I want them to know what is going on in the domain, what do people expect, how do they want to be supported. [...] Starting next year, everyone has an on-site experience day at one of our customers. With the goal to come up with some kind of innovation.”

4.7.4.5 Technology

Web-services Communication components are not that volatile, only if a real new WSDL version is required. So, putting common web-services in components is a good best practice, since they require relative few maintenance. The real problems happen with interpretation of data, which mostly is product specific.

Default behavior Some simple tip for default behavior of making some functionality generic available:

“If you edit code other people are using, making it so that by default nothing changes is a safe best practice.”

4.7.4.6 Testing

At unit B there are not much unit tests. There are some unit tests for database transactions, but unit B mostly relies on regression testing and whatever encountered during development. They are planning to use Selenium tests for the front-end, but this only works for products which are not much actively under development since Selenium tests are a lot of work to maintain. There is no testing on web-services, but there is some testing on data communication since third parties may suddenly change their interpretation of a standard or protocol.

Another employee states that components usually only are tested from the perspective from projects. This means in practice that product-specific functionality is tested, but components are only tested implicitly. Sometimes, for example a communication components, have dedicated tests, but this is rare. Yet another employee states the following regarding testing:

“My experience is that when a unit test fails, people directly act upon it so it really has a function. But my experience is also that the product quality is improving constantly, without us really needing to invest in it. [...] For some errors it is ok that they exist. You should only judge the impact of errors which actually occur. An occasional incident which requires a few extra days to fix then is acceptable.”

Regarding testing frameworks:

“If all teams use different testing frameworks, you cant use their tests because they wont run in your environment ”

Regarding testing on shared components:

“You cant test for modifications on a generic component for all applications. This will require you to test for all products on the platform. Without unit tests, you cant just set up 5 different products and test it yourself.”

4.7.5 Summary unit B

Below a summary of the issues, best practices and desires from unit B.

4.7.5.1 Issues

BTOPP-element	Issue	Associated risks
Business	Different domains Volatile market	14, 17, 18 -

BTOPP-element	Issue	Associated risks
Organization	Management pressure	7, 5
	Justifying updates	13, 16
	Responsible person	26, 27
Process	Knowledge about components	13, 18, 19, 25
	Incorporating components	6, 7, 25
	Refactoring	17, 25
	Justifying updates	13, 16
	Cross-cutting concerns	19, 15
	Testing	10, 18, 19
	Vision of components	12, 13, 14
People	Interests	-
	Mental barrier	26
	Domain knowledge	-
Technology	Component configuration	20, 22
	Communication standards	19
	Splitting components	15
	Email	26

Table 4.5: *Interview results: issues unit B***4.7.5.2 Solutions or best practices**

BTOPP-element	Solution/best practice	Associated risks
Business	Transparency	2, 8, 18
	Future vision	13, 14
	Benefits	6, 25
Organization	Other units	13, 15, 17, 18, 19
	Gatekeeper	13, 15, 26, 28
	Cost sharing	8, 18
Process	Incorporating components	6, 7, 25
	Emails	26
	Development cycle	12, 13, 14
	User feedback	2, 8, 18
	Responsibility	-
	Other units	-
	Product planning	-
	Release cycle	-
Code quality time	-	
People	Responsibility	28
Technology	Web-services	
	Default behavior	
	Testing	10, 18, 19

Table 4.6: *Interview results: solutions or best practices unit B***4.7.5.3 Desires**

BTOPP-element	Desire	Associated risks
Business	A general vision as to how to structure and design packages.	17, 25
Organization	<ul style="list-style-type: none"> - Clarity about responsibilities - Physically located close to other users of your code - Read and write-access on the repositories for everyone, but with adequate quality control - A contact person for certain components 	10, 13, 17, 26 10, 11, 13, 15, 26 13, 15, 17, 18, 19 26, 27, 28
Process	<ul style="list-style-type: none"> - Actively spot platform opportunities - More refactoring on components - More knowledge sharing on technical level - More communication about updates - More unit tests and more knowledge as to how to structure and approach testing - Knowing about upcoming changes more in advance 	25 20, 25 13, 25 10, 17, 19, 25 19 10, 17, 19, 25
People	More curiosity for new or updated technologies	-
Technology	Focus more on modular units of functionality instead of dozens of components	25

Table 4.7: *Interview results: desires unit B*

4.7.6 Unit C

4.7.6.1 Business

Scope of the work For one product a contract is signed for 3 months. A technical design then is made and broken down into smaller pieces of work. The developers make the technical designs, the functional design are made by analysts.

Product vision The interviewee states that customers often don't have an elaborated vision for their products, so what happens is that Unit C develops a vision based on the basic requirements they get from their customers. The interviewee on this:

“In the end customers don't supply us with that much feedback, because customers don't really know what they want. They can come up with some requirements, but we fill in the blanks. But what you want to prevent is that a) they think they get functionality forced down their throat and b) that they are thinking to get functionality which integrates poorly because we made it so generic. [...] We work out the larger picture, the vision.”

Breathing space and resource allocation In order to be able to conduct some refactoring for reuse, investing in designing for reuse in advance or other reuse activities, there has to be room in the planning, resources must be allocated.

“I think that some breathing space has to be created to realize reuse activities. Because of us having a lot of deadlines now, we don't really have the time to tackle these things.”

“In the end, the planning appears to be ill-estimated. You always need additional time to tackle unforeseen issues. You end up spending less time on this than actually should be the case.”

So, since the scope of the work is rather short, there are always deadlines. If you then have the intention to do some additional work to make components available for others, you only increase your technical debt.

“This is always the dilemma with projects and components. You always start on something for a component and at one moment a deadline comes up and you end up not finishing up the work you wanted to do for the component.”

Just allocating more resources is not that simple. You impact other teams in their planning and you need support from the upper management, which is not easy to get if the direct benefit is clearly there.

“Big projects require the approach of an internal project. Often now, somebody needs something for a component and start building it himself. But in an internal project there are more people involved whom benefit from it. [...] but such projects do not happen, because of the deadlines and the fact that in the past this always have been done from the separate product teams. [...] The directors of Care, JeugdZorg and Prototops should come together and say, OK, lets free up 2 weeks to tackle this components across the projects.”

“I have functional demands, I want a stable and fast product. But developers say, yes but with the components we have now we can't do that, so we need to rewrite. Then I say I want some additional functionality. So, then it has to become an internal project where we have to involve people from other teams, resulting in pressure on other deadlines. That's what happens now. [...] What you need is reserving resources in advance for these kind of activities.”

4.7.6.2 Organization

Of interest At unit C, knowledge about components is concentrated around a few people. This is simply because not everyone is interested in components; they don't need it for their job.

“Not everyone is required to work with all other component. A component is used intensively by a distinct group of developers. ”

Responsible person When developing on a shared codebase, you often want to consult with peers for advise on some modification. However, how do you know who to go to? Does that person even have time? The most obvious solution then would be to have a single person responsible for one or more component. As the interviewee puts it:

“With bigger components I'd like to have a leader of the component. A component manager who keeps track of who is working on what component. If you then are updating something you should first come to him and discuss what the impact and risks are.”

Sharing between units The interviewee explains how in practice sharing of code actually works between business units. From the outside it seems units share code, but in reality the code may be similar, but it is located and developed on different locations and never merged together.

“I don’t know exactly what units use our code, but [another unit] uses the form component, [another unit] also. And I think they probably also use the patient record component. The communication component is I think also used by [another unit]. [...] But this is not strange, since they are all spin-offs from the same unit, so they have a large overlap in codebase. [...] For the form component they all have their own branch, so they work on their version. [...] They never pull in updates from us.”

To tackle the responsibility problem, he argues for a component core-team:

“About the whole component story, I don’t know how it is in the world of open-source, but there you often have a core group who oversees the development contributions made by others. I think that is a very interesting structure for the components. A core team and everyone can contribute. The core team than decides what comes in the shared codebase.”

4.7.6.3 Process

Development process The interviewee on the development approach at unit C:

“The development approach is not really that structured. What I try to aim at is to start with an analysis phase with a prototype, then create a design and then start developing. You then break down the work into smaller pieces, pick a piece, create a functional design for it and continue.”

Shared stakes Unit C shares a number of components with unit B. Both units are depending on these components, hence they have a mutual interest in keeping them stable. The interviewee on this:

“For [some project] we have in our unit, together with unit B, the patient record components. Also we have components like patients, employees, organization structure and the communication module. And of course the form component, which unit B is going to use. And we are going to add a new process component.”

Email Communication about updates is done via email or walking around. But, as the interviewee point out, it is the responsibility of the developers to decide what to communicate and to whom:

“Differs per developer if an email is send regarding an update. I leave that to them. The developer should judge the situation, he should look up who has been modifying the code and send those persons and email. Or he could go to them and ask who is using the component and to what extend it can be modified.”

Own responsibility As with the communication of updates, initiating some refactoring or initiating up-front development for reuse relies on bottom-up initiatives:

“For refactoring you just need to make time. I often say to my developers, if something needs refactoring, just do it.”

And also with respect to writing unit tests, the judgment call lays with the developers:

“I expect from a developer that they write a unit test for everything they build. Yes as much as possible, I don’t think it happens that way, but as much as possible. And of course we do a lot of functional testing using regression testing.”

4.7.6.4 People

No interview result here for this unit.

4.7.6.5 Technology

Testing Testing is done by automated unit tests which are automatically invoked by the build server. Besides unit tests, prior to a release regression testing is done following a test script. Some web-services are also tested using unit tests.

4.7.7 Summary unit C

4.7.7.1 Issues

BTOPP-element	Issue	Associated risks
Business	- No breathing room for reuse activities	3, 6, 7, 25
Organization	-	
Process	- Plannings are ill-estimated - Effort of incorporating some component difficult to estimate - When sharing code across units, code is often branched, but never merged back	3, 6 25 15, 16
People	Components are not of interest for everyone	12, 13
Technology	-	

Table 4.8: *Interview results: issues unit C*

4.7.7.2 Solutions

BTOPP-element	Solution/best practice	Associated risks
Business	- Develop your own product vision	8, 14, 18
Organization		
Process	- When sharing code across units, work on your own branch - With respect to communicating changes and writing unit tests for components, that it the responsibility of the developers involved	17, 18 26

BTOPP-element	Solution/best practice	Associated risks
	- Components and web-services tested using unit tests and regression tests	19
People	-	-
Technology	-	-

Table 4.9: *Interview results: solutions unit C*

4.7.7.3 Desires

BTOPP-element	Desire	Associated risks
Business	- Support from the top for resource allocation for reuse activities	6, 7, 8
Organization	- Component-core team	26, 27, 10, 13, 17, 19
	- Component leaders	26, 27, 28
Process	-	-
People	-	-
Technology	-	-

Table 4.10: *Interview results: desires unit C*

§ 4.8 CONCLUSION

In this chapter we give analyzed issues, solutions/best practices and desires from 3 business units at Topicus with respect to working on a shared codebase and dealing with changing requirements. The results as presented in this chapter answers the second research question of this study: *What approaches or techniques can be used to mitigate risks in an agile shared codebase environment better?*. In the next chapters we will use these results to find suitable approaches to implement at FinCare.

- Chapter 5 -

Casestudy: risks at FinCare

§ 5.1 INTRODUCTION

At this moment we have identified the risks of a shared codebase environment with respect to changing requirements (chapter 3) and we have learned how other units at Topicus work with a shared codebase and what problems they encounter (chapter 4). The next step is to identify what risks are relevant for FinCare, which is the third research question: *What are the relevant risks for FinCare?*. In this chapter we answer this question by doing the following:

1. Conduct a bottom-up analysis of the codebase. During the interviews at Topicus we learned that assessing the complexity and nature of the codebase based just on the tacit knowledge of employees from interviews is very difficult. In order to draw conclusions on a technical level, a more detailed analysis is required. The results can be found in section 5.2.
2. Make a short list of the most relevant risks. This is done based on the knowledge gained from the bottom-up codebase analysis, the experience gained during our period at FinCare the last months and informal talks with employees at FinCare and based on feedback talks with the supervisors of this study. The result can be found in section 5.3.

§ 5.2 BOTTOM-UP CODEBASE ANALYSIS

To find out what risks, issues and approaches are relevant for FinCare we conducted a case study on the codebase of FinCare with the FinCareClaim SaaS project as a focal point. In this chapter we discuss the organization, goals and shared codebase underlying this project. An introduction of the case can be found in section 1.3.

5.2.1 FinCare Products

FinCare currently has 4 main projects: CarecoSoft Lite, FinCareAnalysis, FinCareCalc and FinCareClaim. the first 3 are products, the last one is a platform consisting of 4 products.

FinCareAnalysis FinCareAnalysis is an analysis product for health care providers to analyze claims and their relation with the conducted work at a health care facility in detail. The application gives insight in the complex structure of treatments and the distribution of the financial claims over the internal organization of the health care provider.

FinCareCalc FinCareCalc is a back-office application for health care insurance providers to process health care claims. The application checks incoming claims for insurance conditions and automatic auditing procedures.

FinCareClaim FinCareClaim is a SaaS platform for health care providers. The platform supports the claim process for health care providers by means of a highly customizable process which supports automatic checks and controls, status overviews and audit reports. The platform currently knows 4 products: FinCareAlpha, MedicoSoft, CarecoSoft and FinCareClaim.

CarecoSoft Lite CarecoSoft Lite is a web-portal for general practitioners (GP's) which in essence does the same as CarecoSoft, but CarecoSoft Lite is only a web-interface, the back-office is the responsibility of a third-party.

5.2.2 Teams and responsibilities

At FinCare there are 22 employees. Of these 22 employees, there is 1 director, 1 HR/PR employee, 2 GUI-designers, 1 overall senior developer and 1 BI-engineer. The remaining 16 employees can roughly be distributed as follows: FinCareAnalysis (4), FinCareCalc (2), FinCareClaim (7), CarecoSoft Lite (3). Every team has a team leader, but the FinCareClaim platform has two main responsible persons, one for MedicoSoft and FinCareClaim, one for FinCareAlpha and CarecoSoft. However, in the FinCareClaim SaaS project the roles are not that strict, the work is distributed depending on who has time and the expertise for a particular aspect. The GUI-designers and BI-engineer move from team to team depending on where they are needed. The senior developer can be seen as the gatekeeper of the central codebase. He is responsible for maintaining the central packages, he helps in complicated implementation issues and in general is a contact point for technical questions from the developers. The roles in the teams are also not that strict, team leaders do customer acquisitions and develop code, analysts can make work break downs give demo's and write code as well. The roles are flexible, everyone brings in their own expertise and has the freedom to explore in other areas.

5.2.3 Development process

Topicus and also FinCare develop using an development style based on agile practices. FinCare mostly works with fixed-price contracts. In such a contract a number of maximal billable hours for a particular type of work is signed up-front and billed afterward. These items may include for example 'innovation', 'maintenance', 'support', 'implementation costs' or 'development'. Depending on the customer and the life-cycle of the product the items may differ. For example, for CarecoSoft Lite there is little real development and more maintenance work. For FinCareAnalysis and FinCareClaim SaaS this is different, here FinCare is making an investment in developing the product and releasing it to customers via tailored priced contracts. Depending on the wishes of the individual customers, addition implementation effort is done and negotiated for each individual contract.

This means that for new features requirements are specified in financial terms up-front and signed by both parties. In traditional agile development methodologies, requirements are iteratively specified in close cooperation between customer and software supplier. While close customer contact is something which FinCare values, iterative specification of requirements is not the basis and more done in the longer run in terms of months rather than weeks. During development small issues or requests may be included, but the feature planning stays mostly untouched.

At FinCare no agile practice is followed by the book, rather a number of agile practices are incorporated. Scrum-boards are utilized, but no planning poker. Releases are planned using sprints, but only for production releases, not for internal development. A daily stand-up meeting is not required, because in the teams everyone has a good idea of what needs to be done. No item back-log with user stories is used, but rather a feature list with rough functional specifications. So Scrum is not really used, but neither is development done ad-hoc. It sort of follows its course and team leaders can fill in the blanks as they see fit.

Bug reporting and planning is done in Mantis and Redmine. Teams usually have a weekly meeting to make a work break-down for the coming period and discuss the previous period. Releases are done in terms of 4-6 weeks. Functional designs are only made for the larger features, but only if there seems to be a necessity to do so. Technical designs are very sporadically made. Having a release-cycle of test to accept to live is common, but the scope and timing of the releases may differ. After a release sometimes a demo is given, but not always. Feedback from the customers always comes after a production release, sometimes after an acceptance release, but it depends of the feedback given face-to-face or via for example phone or email.

5.2.4 Architecture of FinCareClaim

We have drawn a high-level architecture of the FinCareClaim platform in Von-Neumann notation which can be found in Figure 5.1. As said before the FinCareClaim platform consists of 4 applications: *FinCareClaim*, *FinCareAlpha*, *CarecoSoft* and *MedicoSoft*. Each application consists of 3 main parts in a MVC (Model-View-Controller) layout. Each application has a webapplication, a service and a datamodel. Through the webapplication customers can access dataviews, download invoices and audit reports and initiate a number of processes. In the architecture this are the basic functions as modeled at the web application layer.

In the figure a number of modules are drawn: GrouperModule, FactuurModule, CertificatenModule, AccessControlModule, VecozoModule, COVModule, VektisModule, ExportModule, ZorgeenhedenModule. It was indicated that these modules are shared among all applications. Some support communication with external systems as indicated by arrows drawn outside the FinCareClaim-platform scope.

In the center of the diagram a *control process* box is drawn which depicts the core functionality of the FinCareClaim platform. The core business of the FinCareClaim platform is to process health care claims. These claims can come in from multiple sources and trigger a process. A process is a sequence of events which are chained in a tree-like structure, since some event may invoke different branches in the process. Modeling a process is done in the *WorkflowManager* where using *events* as building blocks some process is created. Events may also be triggers which 'listen' for some condition and are triggered when the condition is met.

Events may be anything, for example data conversion, control procedures, generating reports. It all depends on the type of declaration, the customer or the date. The events are generic little applications written in plain C#.NET.

The last block in the diagram is the *Queue*. The queue is used when for example some event is invoked to generate a batch of invoices, which may be triggered from the web application.

5.2.5 Technical architecture

All applications are written in C#.NET. The source code of the applications is managed by Mercurial subversion control. The repository top-level is illustrated in Table 5.1. A .NET application is called a compiled solution and a solution is a collection of assemblies or .NET projects. Every assembly can include packages, which can be compiled solutions outside the scope of the current solution or external packages and references to other assemblies in the same

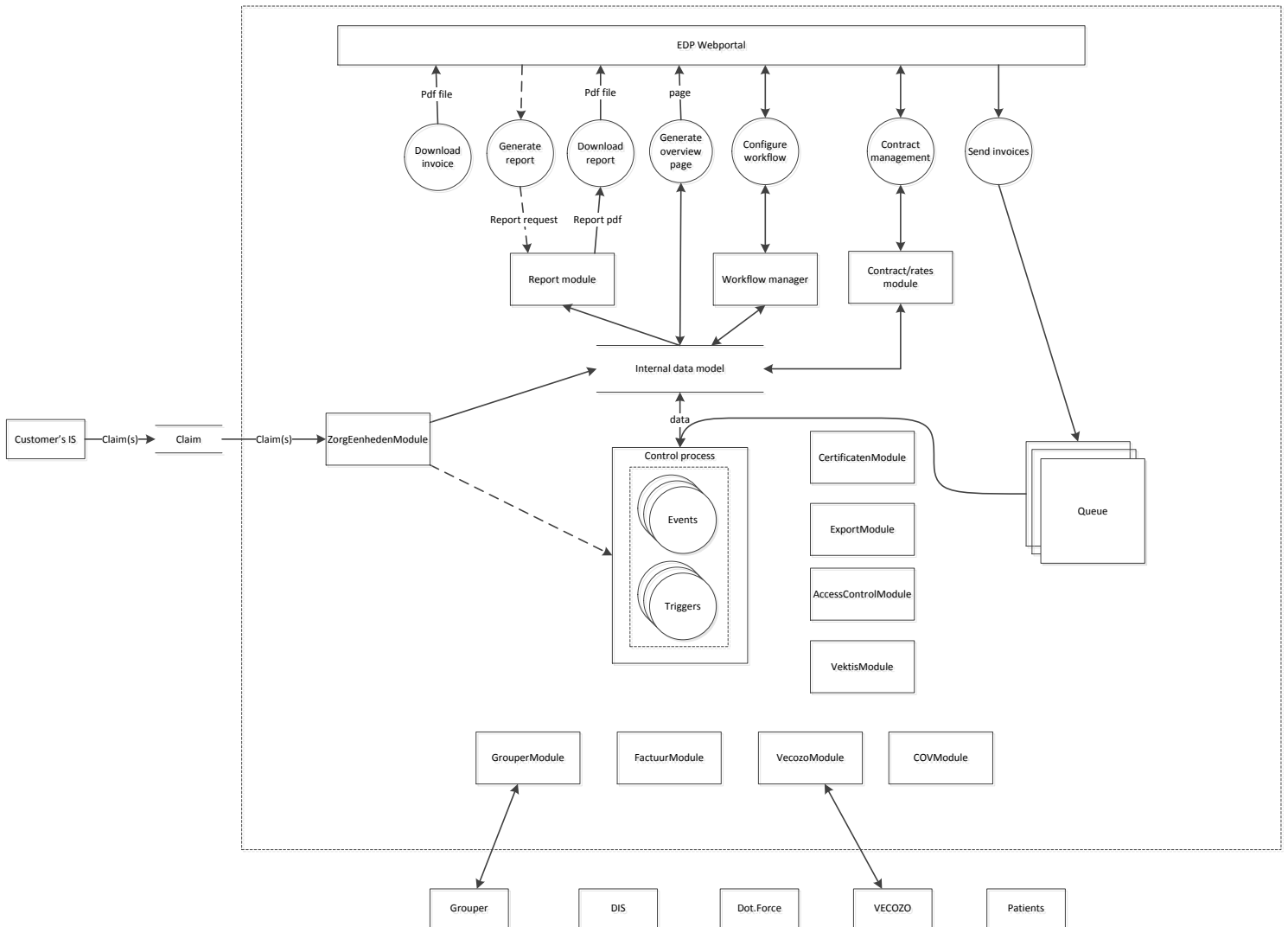


Figure 5.1: High-level architecture of the FinCareClaim platform

solution. Having an assembly in the scope of the current solution means that you have direct access to the source code and can compile it. A package on the other hand is outside of the scope of the solution and hence cannot be directly edited. If we look inside the repository of the FinCareClaim platform we see a list of 69 folders, most of them representing an assembly. A snapshot is given in Table 5.2.

The *declaratiegeneratie*, *FinCareAnalysis*, *CarecoSoft Lite* and *force.facts* repositories all are built up in the same way.

5.2.6 Modules, components, packages and libraries

For this study we are interested in the shared codebase. From the repository structure alone, we cannot directly see what is what. In the *declaratiegeneratie* repository for example, 4 different applications exist. Of which assemblies are they composed? What assemblies do they share? Is there any sharing between the assemblies of FinCareClaim and the other applications outside the *declaratiegeneratie* repository?

We learned that an assembly can be recognized by the *.csproj files in the folder, which

Repository	Project
cubseservice	BI cubseservice
declaratiegeneratie	FinCareClaim platform
FinCareAnalysis	FinCareAnalysis project
FinCareAnalysis_selenium	FinCareAnalysis selenium tests
force.facts	Reporting dashboard
Medico_selenium	Selenium tests for MedicoSoft
CarecoSoft Lite	CarecoSoft Lite project
CarecoSoft Lite_selenium	Selenium tests for CarecoSoft Lite
CarecoSoft_selenium	Selenium tests for CarecoSoft

Table 5.1: *Mercurial repositories at FinCare*

Repository	Project
cubseservice	BI cubseservice
declaratiegeneratie	FinCareClaim platform (69) Careco.NabetalingOpAchterstandswijken Certificaten Declaratiegeneratie Declaratiegeneratie.CustomConfiguration Declaratiegeneratie.Engine.Bundels [..] NContract.WebService.Stub NControl.Webservice.Stub Releasenotes Scripts Testen WSDLS
FinCareAnalysis	FinCareAnalysis project
FinCareAnalysis_selenium	FinCareAnalysis selenium tests
force.facts	Reporting dashboard
Medico_selenium	Selenium tests for MedicoSoft
CarecoSoft Lite	CarecoSoft Lite project
CarecoSoft Lite_selenium	Selenium tests for CarecoSoft Lite
CarecoSoft_selenium	Selenium tests for CarecoSoft

Table 5.2: *Collapsed FinCareClaim-platform repository*

is a C# project file for Microsoft Visual Studio. Here a number of references exist, a project reference (an assembly in the current solution) and a reference to a package (pre-compiled external solution).

We therefore define a *package* as outside of the active solution located solution and we define a component as a single assembly in the scope of the active solution. A module is an assembly which has a clear, isolated function in or between applications.

5.2.7 Visualizing the repositories

Using Gephi (Bastian et al., 2009) and some self-written small-tools a number of network graphs in the open GEXF file format have been created to visualize the internal structure of the codebase

and allow for detailed analysis of the topology. In Figure 5.2 the complete repository dependency graph is given. In this graph nodes represent all individual components and packages of the five repositories. We gave every component package colors as depicted in Table 5.3.












Repository/type	Color
FinCareClaim platform	
FinCareAnalysis	
CarecoSoft Lite	
Force.facts	
Cubeservice	
NuGet packages	






Table 5.3: *Color mapping*

When giving a first glance at Figure 5.2 it seems that the codebase is a complete mess where everything is entangled. In other words, we don't know what we see here. The nodes have been given a size according to their network degree (sum of in- and outgoing links) and we see that there is a clear clustering of assemblies around a set of approximately 10 packages. Also, all applications seem to have a part which is highly entangled and a part which is more isolated. If we remove the packages and look at the assemblies only, we get the graph as depicted in Figure 5.3.

Here we can see that force.facts assemblies are only included in the FinCareAnalysis project, but between the repositories there are no links anymore. This is hardly surprising, since this is the role of the packages. In Figure 5.4 we grouped the repositories and switched the packages back on.

Now we have a very interesting picture where we can see that there seem to be a group of packages shared by all repositories and packages shared by a subset of the repositories. For further analysis we listed all the packages and mapped them to the repositories. This mapping can be found in Table 5.4.

Package	Degree					
FedZorg.Facturen	18	x				
Common.Logging	6	x	x	x	x	x
Force.Basics.NHibernate	5	x	x	x	x	
Iesi.Collections	5	x	x	x	x	
NHibernate	5	x	x	x	x	
NHibernate.Mapping.Attributes	5	x	x	x	x	
Spring.Aop	5	x	x	x	x	
Spring.CodeConfig	5	x	x	x	x	
Spring.Core	5	x	x	x	x	
Spring.Data	5	x	x	x	x	
Spring.Data.NHibernate32	5	x	x	x	x	
NLog	5	x	x	x	x	x
Force.Basics.NLog	5	x	x	x	x	x
Force.Basics	5	x	x	x	x	
Force.Basics.Workflow.Core	4	x	x	x		
Newtonsoft.Json	4	x	x	x	x	
Force.Basics.NHibernate.Spring	4	x	x	x	x	

Package	Degree					
Quartz	4	x	x	x	x	
Spring.Services	4	x	x	x	x	
NHibernate.Caches.SysCache	4	x	x	x	x	
elmah.corelibrary	4	x	x	x	x	
Force.Basics.Mvc3	4	x	x	x	x	
Spring.Web	4	x	x	x	x	
Spring.Web.Mvc3	4	x	x	x	x	
RhinoMocks	4	x	x	x	x	
Force.Basics.AccessControl.Core	4	x	x	x		
Force.Basics.Workflow.AccessControl.Core	3	x	x	x		
FedZorg.Certificaten	3	x		x		
SharpZipLib	3	x		x		
Common.Logging.NLog20	3	x	x		x	
Atlas	3	x	x		x	
Autofac	3	x	x		x	
elmah	3	x	x	x		
jQuery	3	x	x	x		
jQuery.UI.Combined	3	x	x	x		
Modernizr	3	x	x	x		
T4MVC	3	x	x	x		
FedZorg.Vektis	3	x		x		
Force.Basics.AsciiIo	3	x		x		
FedZorg.Vecozo.Declareren	2	x				
FileHelpers-Stable	2	x	x			
Microsoft.Web.Infrastructure	2	x	x			
Boo	2	x	x			
Boo-Compiler	2	x	x			
FileHelpers	2	x	x			
RhinoDSL	2	x	x			
Rhino-Etl	2	x	x			
FedZorg.Cov	2	x		x		
NHibernate.Logging	2	x		x		
System.Data.SQLite.x64	2	x		x		
Glimpse	2	x		x		
Glimpse.Elmah	2	x		x		
Glimpse.Mvc3	2	x		x		
glimpse-dependencies	2	x		x		
Mvc3Futures	2	x			x	
MvcContrib.Mvc3-ci	2	x			x	
NHibernate.Glimpse	2	x		x		
jQuery.Validation	2	x		x		
NVelocity	2	x		x		
T4MVCExtensions	2	x	x			
jQuery.jqGrid	2	x		x		
Spring.Template.Velocity	2	x		x		
Rx-Main	2	x	x			
Spring.Scheduling.Quartz	2		x	x		
EntityFramework	2		x	x		

Package	Degree					
		x		x		
jQuery.vsdoc	2	x		x		

Table 5.4: Shared packages with degree higher than 1

Before drawing any conclusions we first zoom in into the FinCareClaim platform. The question we have here is what packages and assemblies are shared between the different application inside the platform. The complete codebase dependencies of FinCareClaim are depicted in Figure 5.5. The large dots represent the main components of the individual application. Still, the graph is rather cluttered. So, we used simple regular expressions to group the applications, for MedicoSoft we used for example: $(.*)MedicoSoft(.*)$. The result is the graph as depicted in Figure 5.6.

From the graph we abstracted a list of assemblies and their in- and out-degree which can be found in Table 5.5.

Assembly	In	Out	FClaim	MSoft	CSoft	FAlpha
Declaratiegeneratie	6	0	x	x	x	x
Declaratiegeneratie.Engine.Facturen	4	0	x	x	x	x
Declaratiegeneratie.Export	4	0	x	x	x	x
FedZorg.Facturen	4	0	x	x	x	
Declaratiegeneratie.Engine.Zorgeenheden	4	0	x	x	x	x
Declaratiegeneratie.Engine.Bundels	4	0	x	x	x	x
Declaratiegeneratie.Zorgaanlevering	3	0			x	x
Declaratiegeneratie.Engine.HIS	3	0		x	x	x
Declaratiegeneratie.Web.Area.Beheer	2	0		x	x	
Declaratiegeneratie.Workflow	2	0			x	x
Declaratiegeneratie.Koppeling.NContract	2	0		x	x	
Declaratiegeneratie.AccessControl	2	0			x	x
Declaratiegeneratie.Koppeling.CarecoSoft Lite	2	0			x	
Declaratiegeneratie.Lucene	2	0		x		
Declaratiegeneratie.WebApplication.Area.Testpages	1	0			x	
Declaratiegeneratie.Sntp	1	0			x	
Declaratiegeneratie.CustomConfiguration	1	0			x	
Declaratiegeneratie.Koppeling.Twinfield	1	0			x	
Declaratiegeneratie.Koppelingen.Grouper	1	0	x			
Declaratiegeneratie.Koppeling.SHL	1	0			x	
Declaratiegeneratie.Koppeling.FinCareAnalysis	1	0	x			
Declaratiegeneratie.Zorgaanlevering	0	26			x	
Declaratiegeneratie.Zorgaanlevering.Tests	0	19			x	

Table 5.5: Shared assemblies of FinCareClaim

From the table and the figure we can recognize a number of modules from the high-level architecture (GrouperModule, FactuurModule for example in the form of respectively the Declaratiegeneratie.Koppelingen.Grouper and Declaratiegeneratie.Engine.Facturen assemblies). Since they are assemblies, they may be edited from the active solution. The next question then is, from the components we have found to be part of the shared codebase of the FinCareClaim platform: which have a high chance having impact on another component when modified?

To analyze this we created a small-tool which mines the commit-messages from the Mercurial repositories and by using association rule learning techniques (Han and Kamber, 2006) we tried to find the relation between assemblies. The idea is based on logical coupling (Gall et al., 1998) and item-set calculation for determining traceability links between repository assets (Kagdi et al., 2007).

The result of the association rule analysis is depicted in Figure 5.7. In this figure the bigger the nodes, the larger the number of associations. Still, the graph is a bit cluttered because there

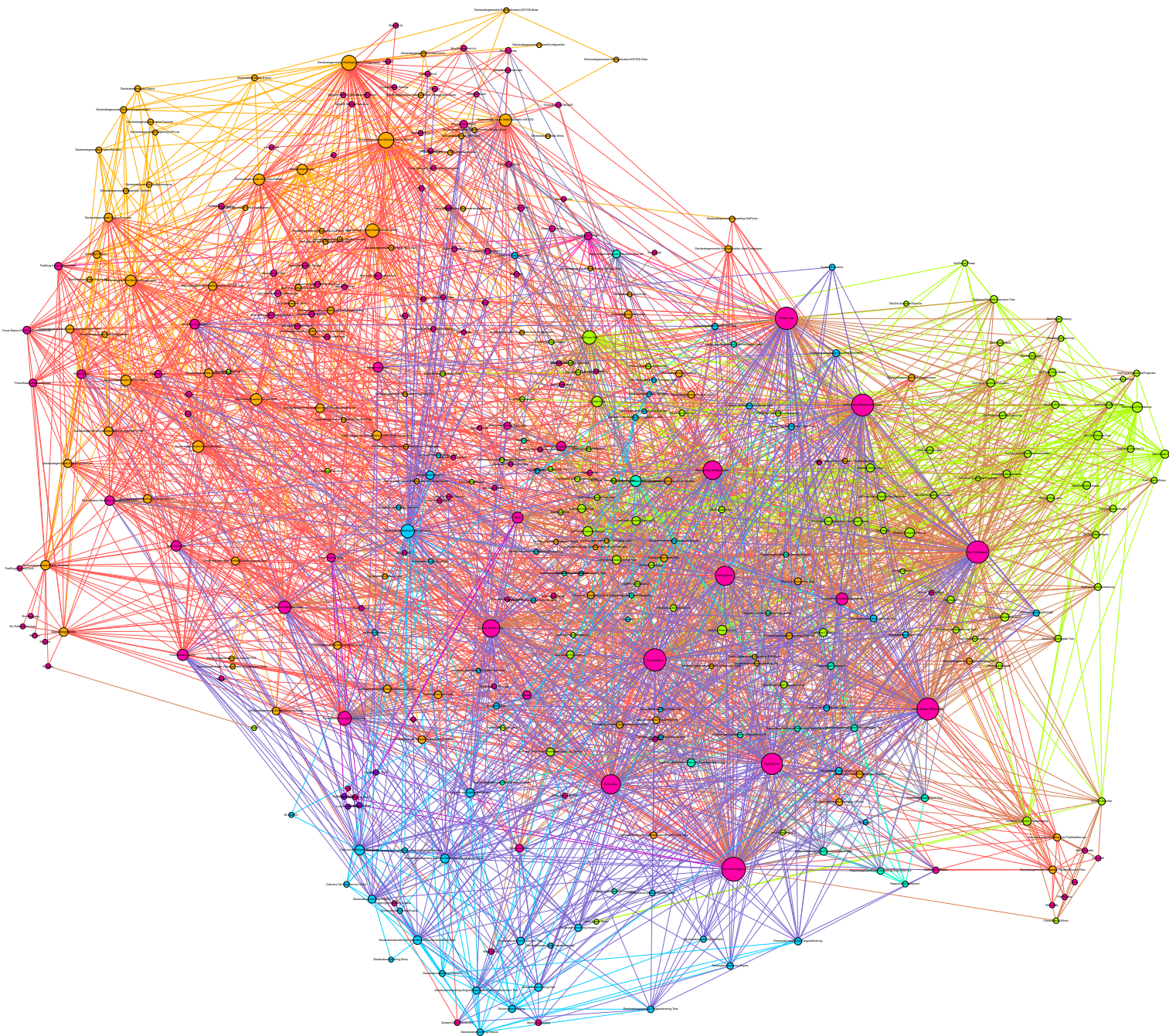


Figure 5.2: Complete dependency graph of codebase at FinCare

is no clear distinction between the application. Therefore we again grouped the assemblies by the web-application, service and data components. The result can be found in Figure 5.8.

So, what do we see in Figure 5.8? The red nodes represent the different applications. The more blue the other nodes, the more they are associated with another component or a set of components when a file of such a component is edited. Also, the thicker the arrow between two nodes, the stronger the association. We can observe a number of things:

- There is a very strong bi-directional association between *CarecoSoft* and *MedicoSoft*.
- The FinCareAlpha application is not visible in the graph. This is because they have a

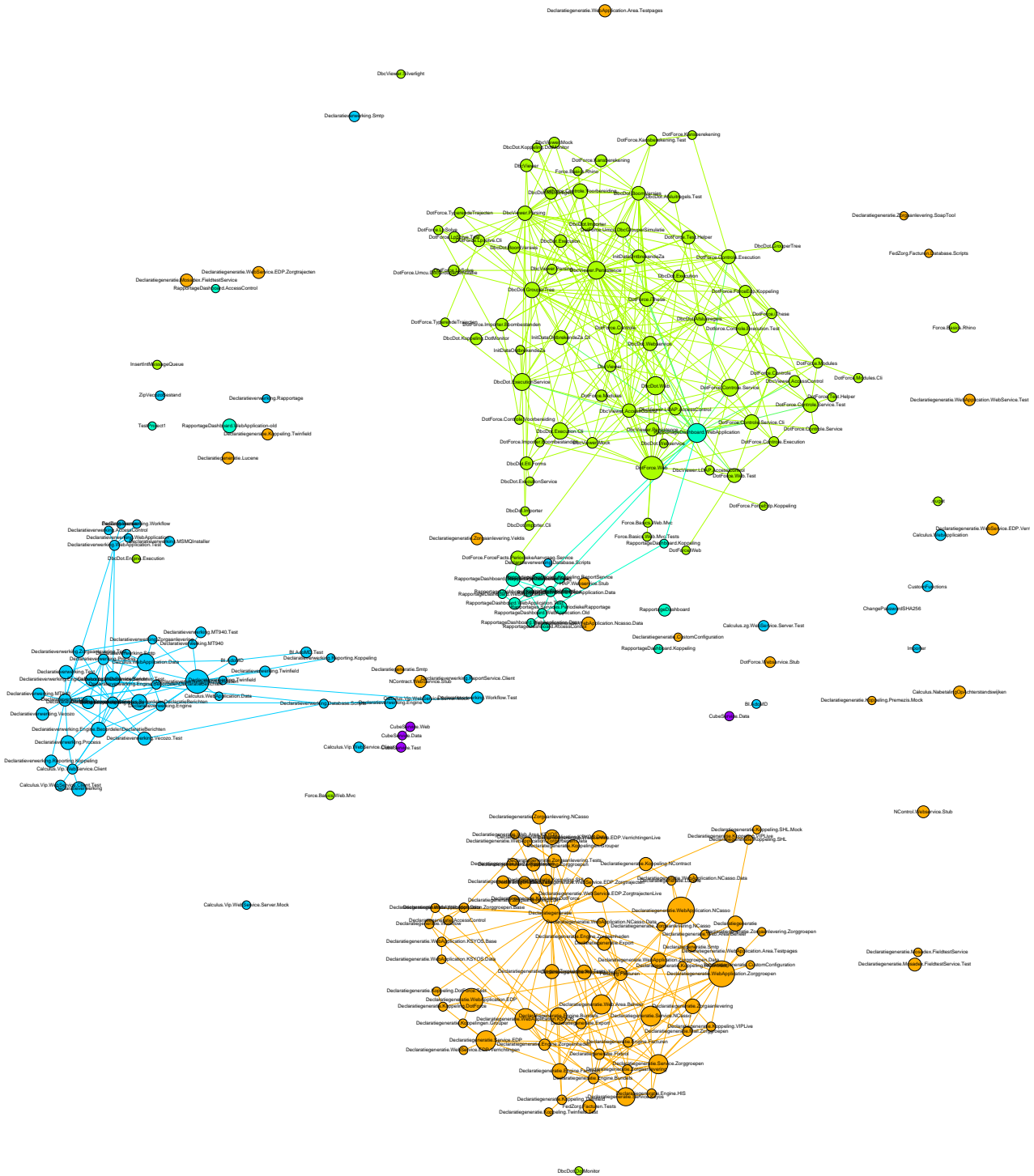


Figure 5.3: Dependencies between assemblies

separate branch in the repository.

- There are a number of assemblies which have a high number of associations:
 - Engine.Facturen
 - Declaratiegeneratie
 - Engine.Bundels

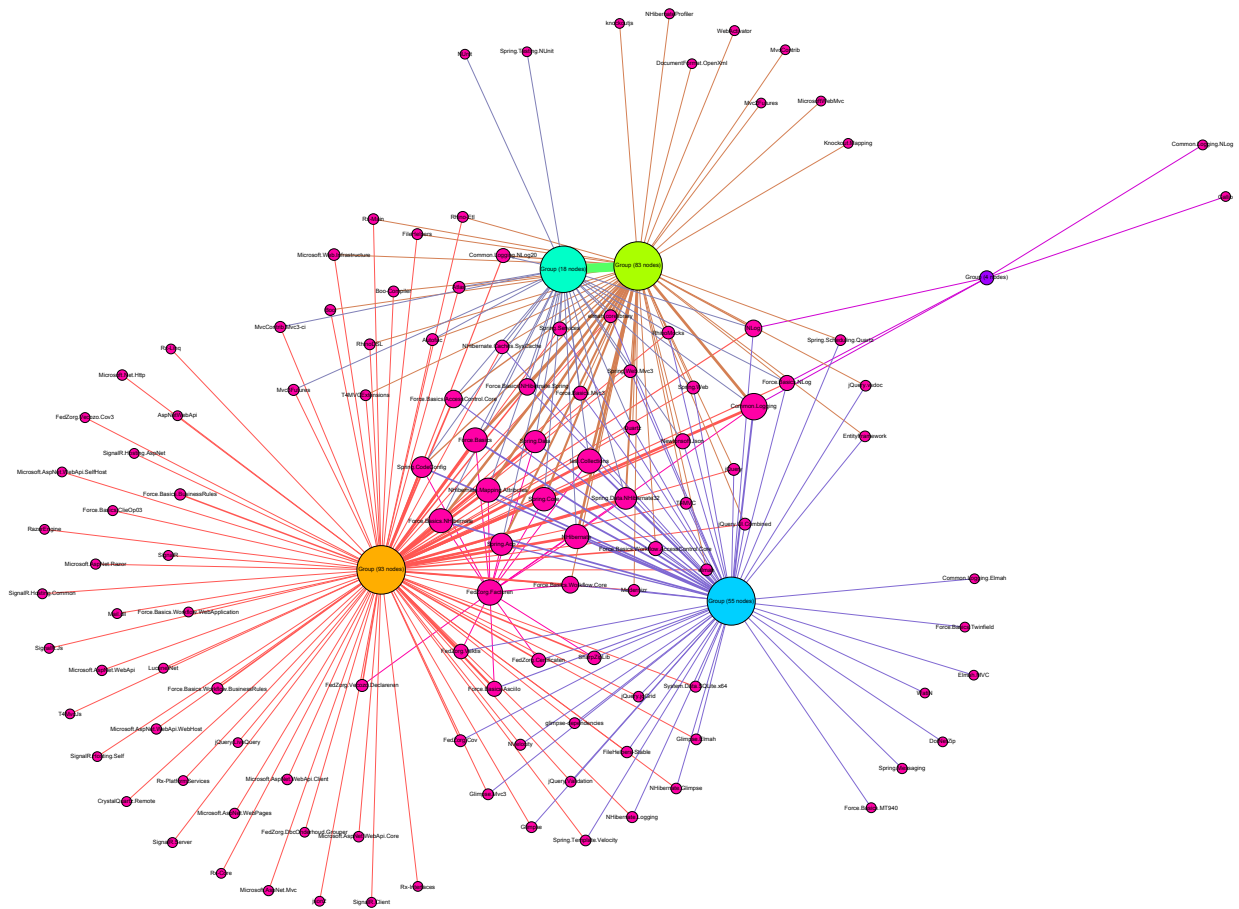


Figure 5.4: *NuGet dependencies between repositories*

- Engine.Zorgeenheden
- Web.Area.Beheer
- Zorgaanlevering

If we go back to Table 5.5 we can see that the 6 components with a high number of associations are used by mostly all applications. Hence, there is a very high change that the other applications are impacted, since the components are highly volatile. The second interesting thing is the strong bi-directional association between *CarecoSoft* and *MedicoSoft*. We created another graph where we only look at commits from 1 year ago and further in the past. The result can be found in Figure 5.9.

5.2.8 Conclusions

From the bottom-up analysis of the codebase we can conclude the following:

- There is a large overlap of components between all applications of FinCare. From all the shared applications, almost all packages are used by applications from the FinCareClaim platform. From the packages a distinction can be made between really the core packages and sub-domain specific packages.
- In the FinCareClaim platform there a number of modules which are tightly coupled to the platform’s applications in terms of both logical and semantic dependencies.

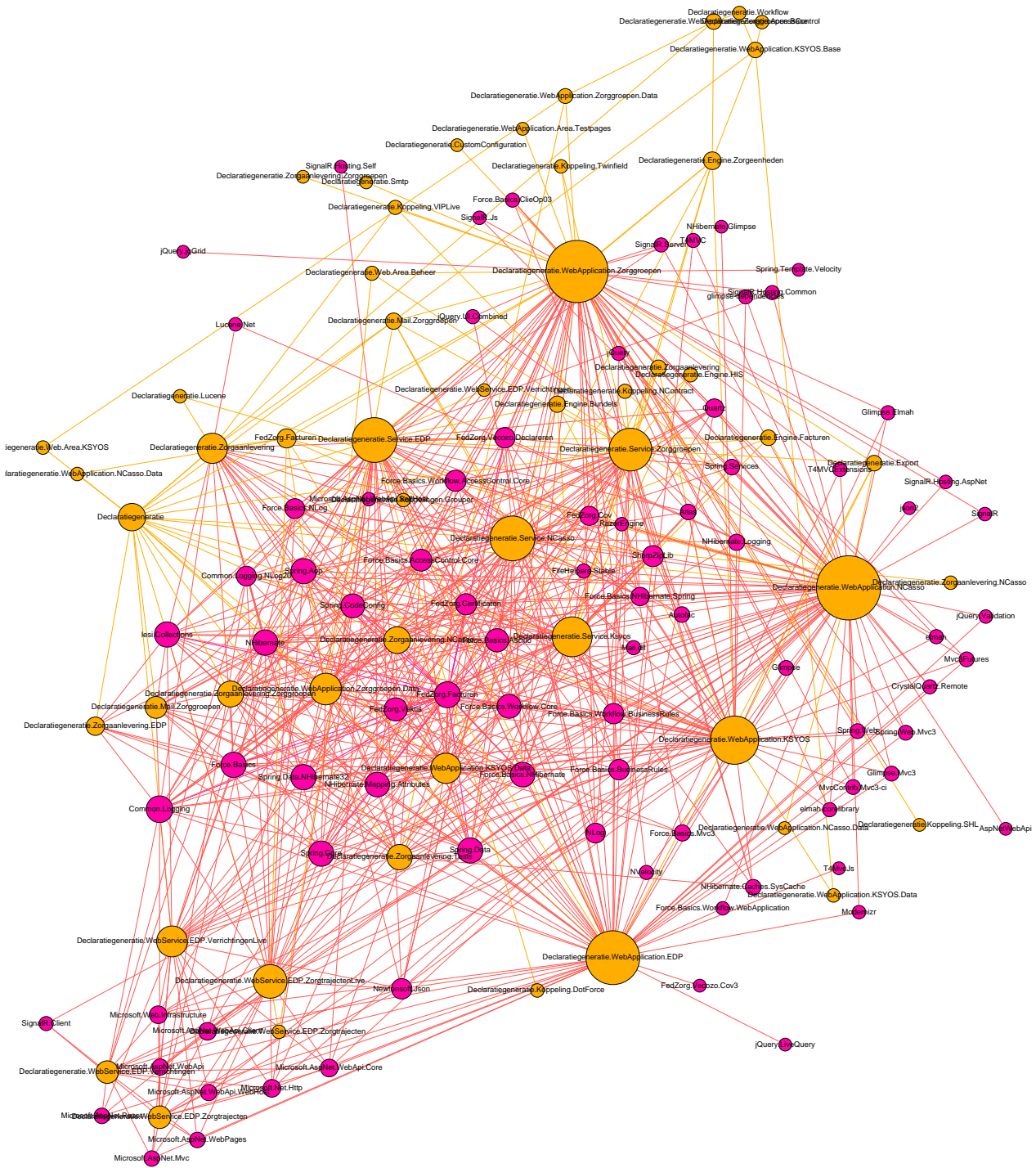


Figure 5.5: Complete codebase dependencies of the FinCareClaim platform

- The MedicoSoft and CarecoSoft applications are tightly coupled in terms of mutual dependencies and logical dependencies between the core assemblies.

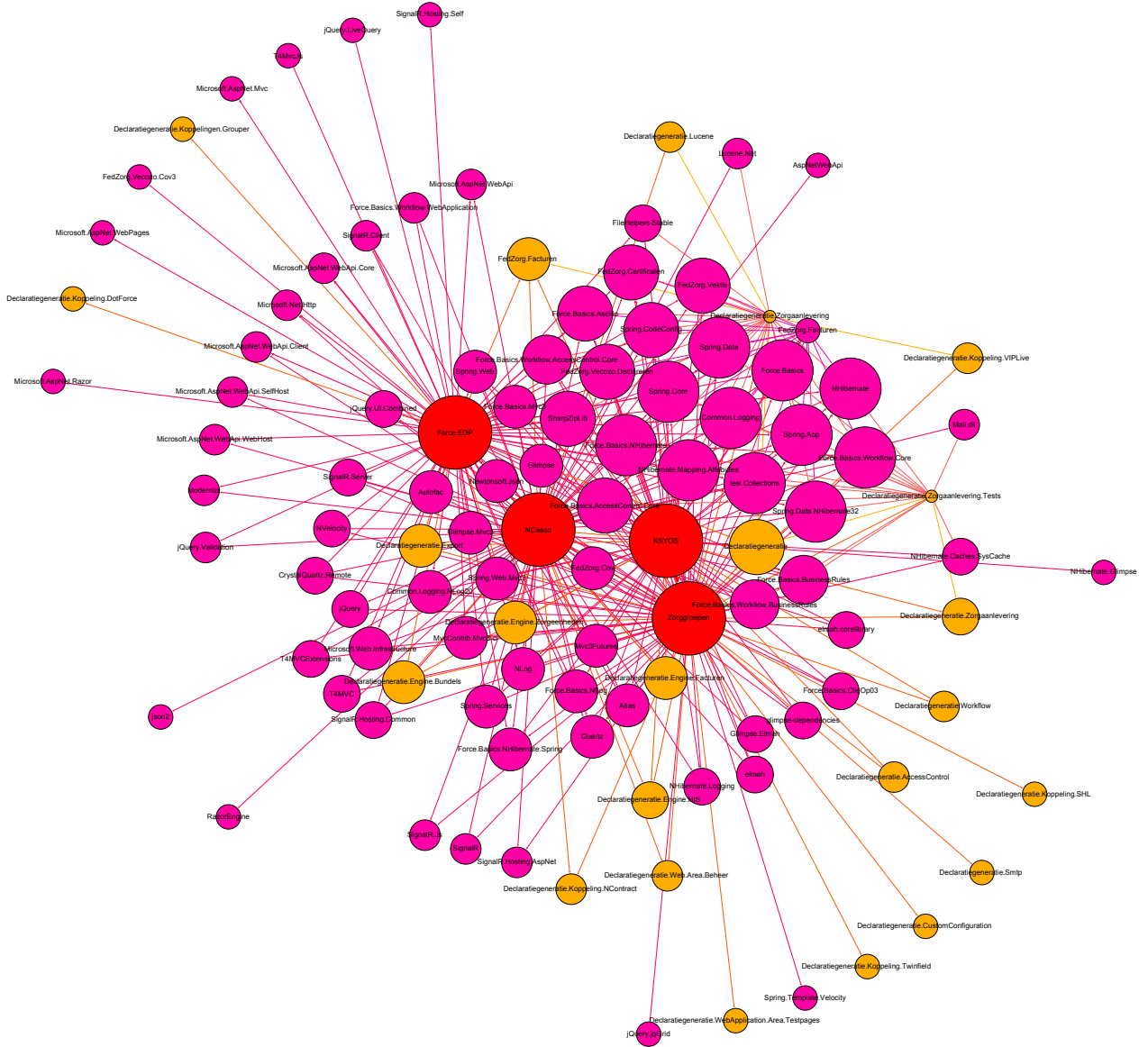


Figure 5.6: *FinCareClaim* platform grouped per application

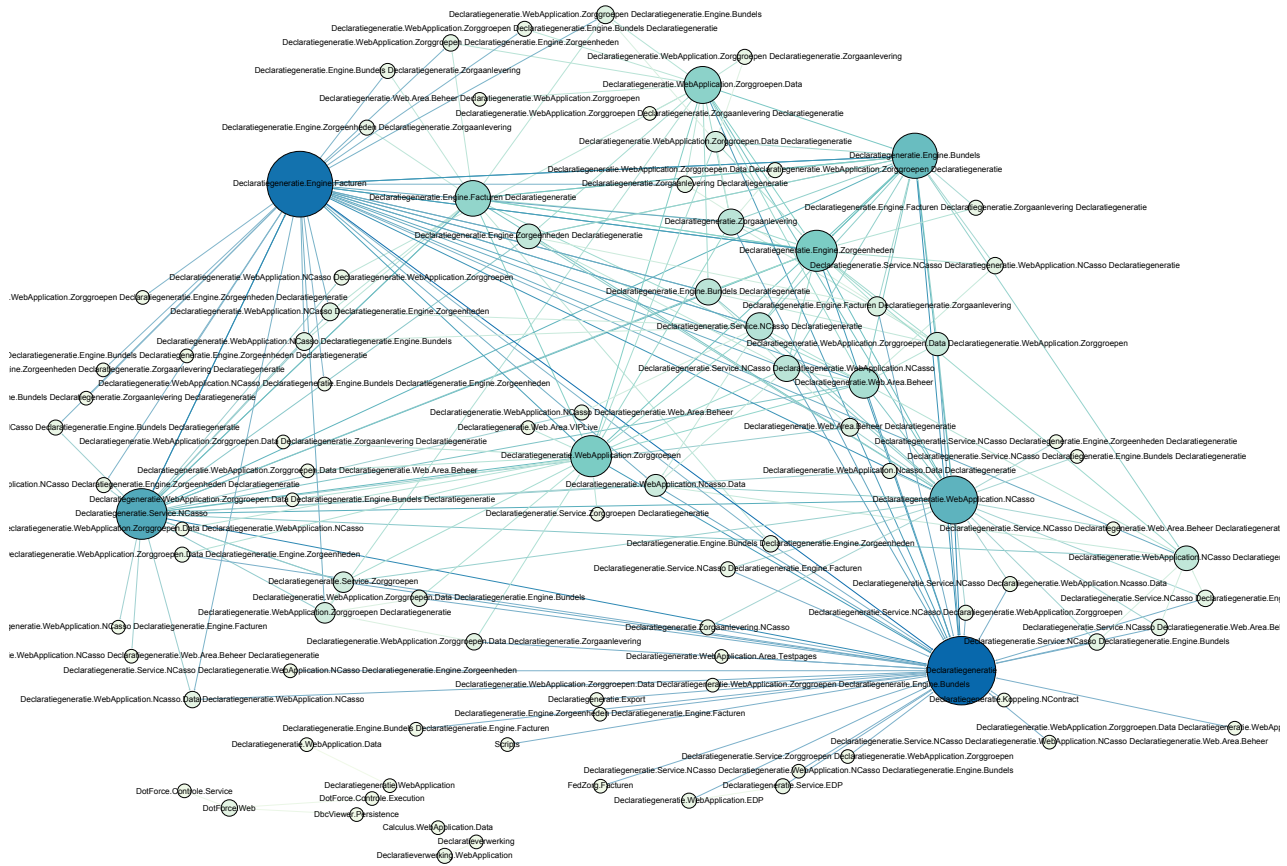


Figure 5.7: Logical coupling of FinCare repository

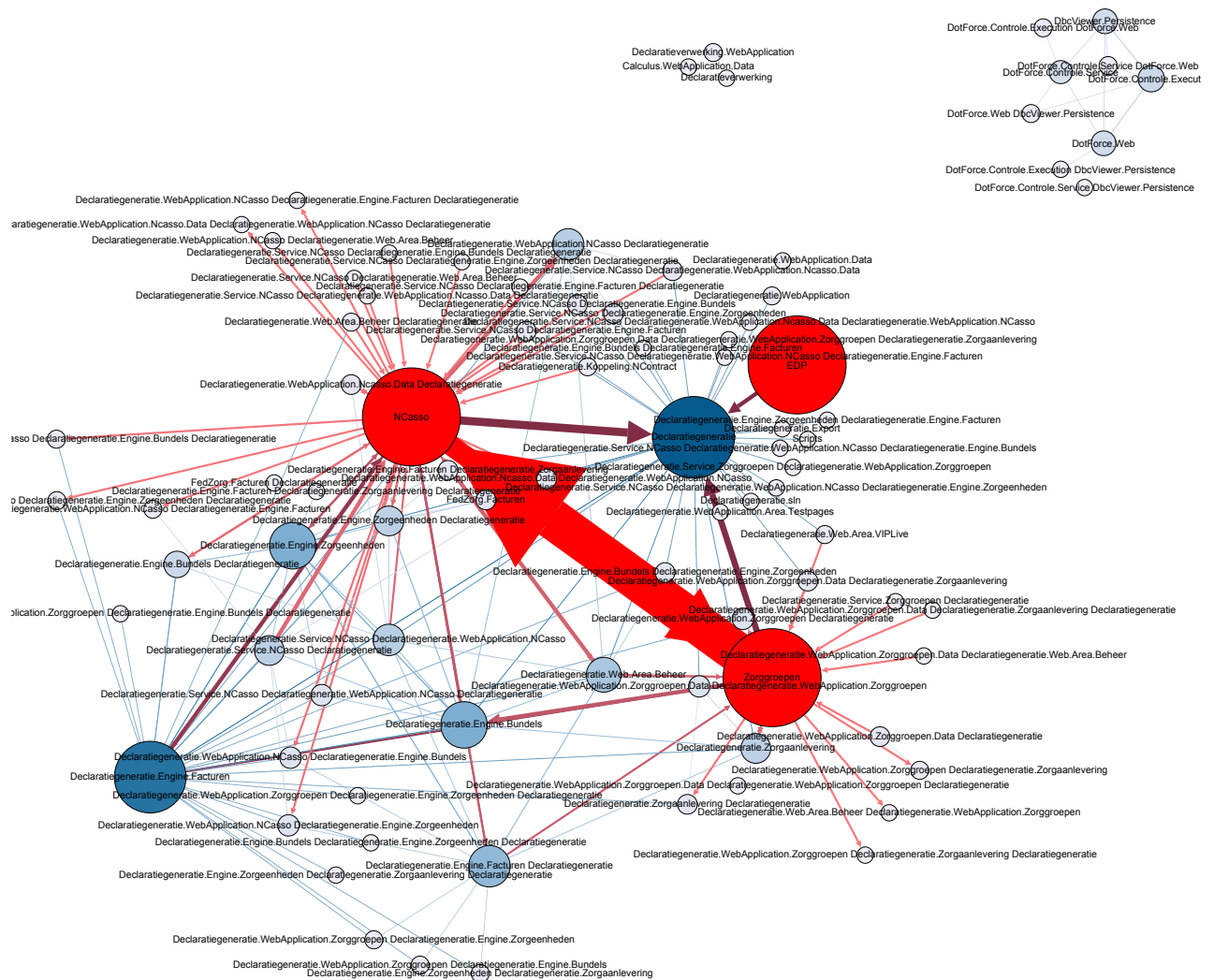


Figure 5.8: Logical coupling of FinCare repository grouped by FinCareClaim applications

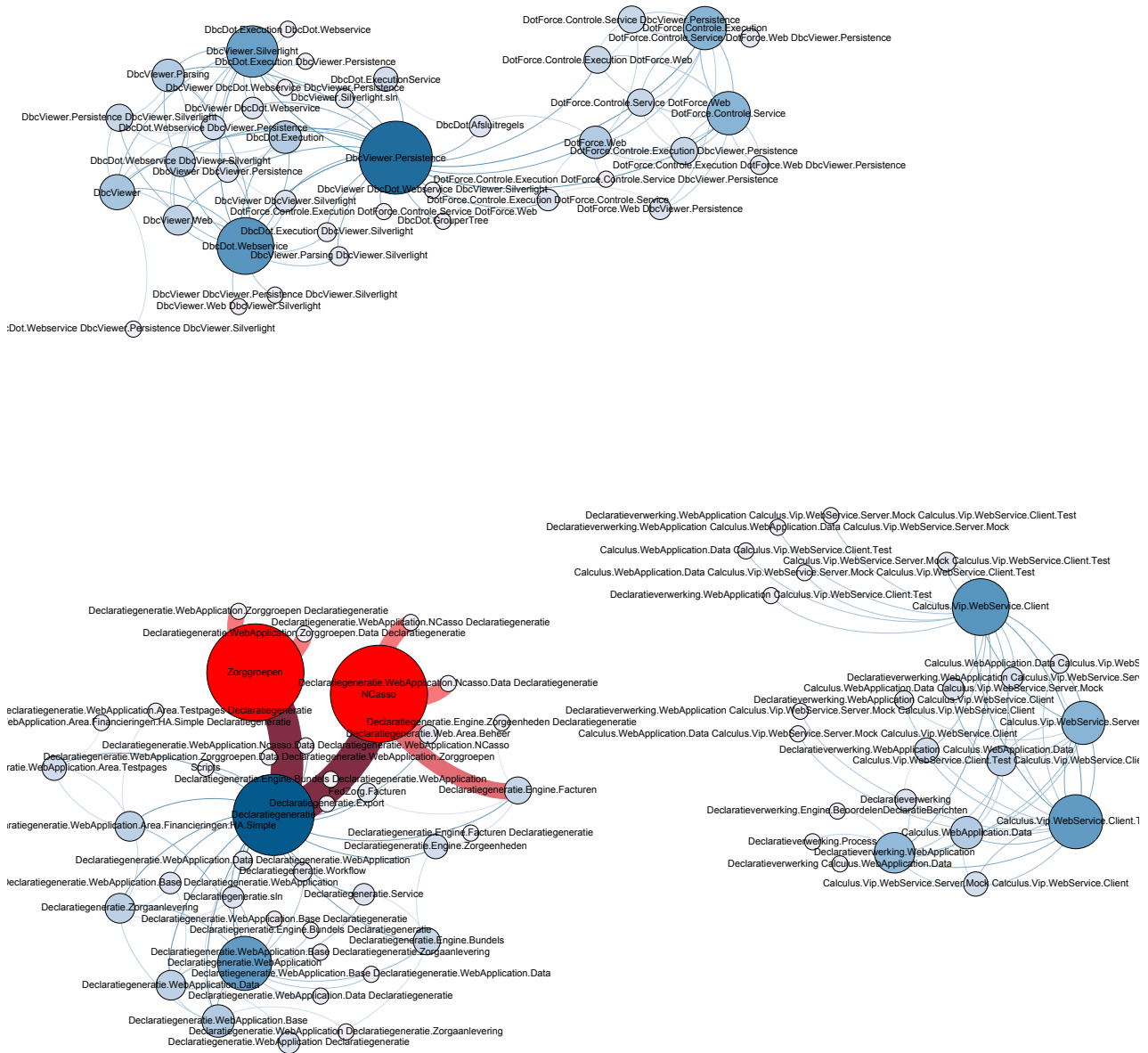


Figure 5.9: Logical coupling older than 1 year ago

§ 5.3 RELEVANT RISKS

5.3.1 Introduction

In this section we discuss the risks from section 3.4 and assess their relevancy. The goal is to make a short list which then can be used to find suitable mitigation approaches. We define relevancy in terms of the dimensions *impact* and *likelihood of occurrence*. We have each dimension a value on a 5-point ordinal scale (very high, high, medium, low and very low).

5.3.2 Discussion of risks

#1 Risk of operating in a volatile market The teams in general have short lines with their customers. This means that when there are market changes, the news and effect ripples very fast to FinCare. Also, a volatile market is the core value proposition for the customers of FinCare. Their products are based around managing complex and changing business rules in the health care domain. However, while it may impact the individual products, it does not affect the shared codebase that much. Rather, the products have a wide variety of configuration options to capture these kind of changes. So the likelihood is very high, but the impact is medium, hence the overall relevancy is high.

Likelihood: Very high. *Impact:* Medium. **Relevancy:** High.

#2 Risk of stakeholder influence The lines with customers are short. Customers communicate frequently with not only the product owners, but also with analysts and sometimes developers. Changes therefore can easily slip into the development cycle this way making the likelihood high. The impact would be high if this caused a lot of additional work to be done which would affect code quality. But at FinCare everyone is smart enough to know how to balance customer relations and code quality. They know when to say no. Nevertheless, the commercial aspect is still very important. Saying yes to some additional feature because of the prospect of strengthening the relation can have impact on other products, and with the current customer relations that is a realistic scenario. We think however that this is managed very well at FinCare, hence the impact is low giving this risk a medium relevancy.

Likelihood: High. *Impact:* Low. **Relevancy:** Medium.

#3 Risk of time-to-market pressure One of the reasons customers like to work with Topicus is that they are fast and flexible. But because of their iterative development approach, there is always a deadline on the horizon where the teams are expected to deliver a new release. For FinCareClaim SaaS new customers are already on the horizon, promises made, so the pressure is high. The impact would be high if for that reason design choices are made which compromises the quality and re-useability of the codebase. Because FinCare mostly works with fixed price contracts, the functionality to implement in the next couple of releases for the large part is fixed. Additional work then always asks for compromises.

Likelihood: High. *Impact:* High. **Relevancy:** High.

#4 Risk of evolving standards The products of FinCare use a number of standards, which may change, but the chances of this happening are very low. If a standard changes, the impact of the change very much depends. The standards with the most impact have the least chance to change. For the rest, the products can deal with a lot of changes with small adjustments or utilize configuration options.

Likelihood: Very low. *Impact:* Medium. **Relevancy:** Low.

#5 Risk of political aspects In the health care domain from time to time laws and regulations may change, which may affect all products at FinCare. For these kind of changes, the flexibility of the configuration of the work-flow manager can capture such changes. While some developments can have a severe impact on the products, the impact is medium for the shared codebase.

Likelihood: High. *Impact:* Medium. **Relevancy:** Medium.

#6 Risk of business philosophy focusing on short-term goals For the frameworks at FinCare there is an attitude to do some investment up-front to have a stable product, but when the first customer(s) come into sight, the focus shifts to delivering the product. For FinCare-Claim SaaS some customers are on the horizon, and the product is parallel being implemented for these customer and for further customers. Having short release cycles and a road-map with a short scope does not stimulate careful platform planning and design for reuse. Concessions on the design, less refactoring, more potential code-smells then can be the result. Less is the risk of loosing the financial investment when the roll-out after the invest-period proves not to be profitable, but since they have a number of good prospects the impact is medium. This can change however if the number of products to deliver increases.

Likelihood: Very high. *Impact:* Medium (now). Becomes more important if the amount of products increases. **Relevancy:** Medium (now).

#7 Risk of business value thinking This risk is relevant if there is an attitude towards reuse and platform design that it should be convenient in both effort and financial terms. At FinCare there is little focus on achieving high levels of reuse, while there may be enough opportunity. This is because framework code refactoring or reuse for other applications than yourself does not directly show any benefit. In that sense, the likelihood for this risk is high for FinCare. The impact is also high, since the number of products to manage only will increase over time. FinCare is attracting a good number of new prospects, hence if no time is spent on designing for reuse it may jeopardize the quality of the codebase in the future.

Likelihood: High. *Impact:* High. **Relevancy:** High.

#8 Risk of prioritizing on mainstream product This risk becomes relevant when there is a clear platform with one mainstream product. The FinCareClaim platform not really has a main-stream product, so in that sense the risk has a low relevancy. However, from the perspective of all the packages combined, the FinCareClaim platform as a whole compared to the other products has a relative large stake in the shared codebase. The more important FinCareClaim becomes financially for FinCare, the more there is chance that fixes, updates or other modifications are done from a perspective of FinCareClaim, but which affect the other products.

Likelihood: Low. *Impact:* Low. **Relevancy:** Low now, increases with financial stakes of the FinCareClaim platform.

#9 Risk of changing the business strategy FinCare has a stable position in the market, but they cannot afford to sit still. However, aggressive marketing is not required. There are a good number of prospects and stable customers. Also, there is little depending on third parties which FinCare has to follow.

Likelihood: Very low. *Impact:* Medium. **Relevancy:** Low.

#10 Risk of reusing immature components The main modules of FinCareClaim have been under development for 2-4 years. They reached a level of maturity where they are reusable and stable. So prematurely reusing them is not an issue. It may be an issue for events, since

they are constantly under change, but this is on such a low-level scale that it is comprehensible. The likelihood of this happening is medium, since modifications on them are made from time to time, but no heavy re-designs. The impact of the external packages on this topic is more difficult to predict, since you then would have to know the ‘maturity’ of all the used packages. While from time to time these packages are updated which can cause some daily frustrations, the impact seems manageable.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

#11 Risk of unclear requirements Because FinCare has relatively small teams (3-5 persons), and because requirements are used only inside teams, lack of clarity about requirements are easily resolved. Also the scope of requirements is small since the iterations are short, hence there is little material to have lack of clarity about. So therefore the relevancy of this risk of low.

Likelihood: Low. *Impact:* Low. **Relevancy:** Low.

#12 Risk of different interpretations of artifacts Level of collaboration outside the business unit is relatively low. Internally, there of course is plenty of discussing about documents and other artifacts, but uncertainties are easily solved by shouting around or walking to the responsible person next door.

Likelihood: Low. *Impact:* Low. **Relevancy:** Low.

#13 Risk of goal ambiguity While the teams are small, there is no clear component-level documentation as to what sub-systems or packages’ functions are. Also the shared components are managed mostly by a single-person, increasing the risk that the knowledge stays only with the same person. Yet, the problem is easy to resolve by walking to the person who knows more about the component. So, therefore the relevancy is medium.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

#14 Risk of scope widening FinCare’s products are not all equally mature and clients are attracted as much as they can find. Customers are not rejected because a product is not mature yet, but rather because the capacity is not there. Scope widening therefore can be a risk if because of the scope widening new functionality is added without looking at well-designed reuse options. Yet, everyone is capable enough to deal with this and scope widening happens in small steps. So therefore this is a risk of medium relevancy.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

#15 Risk of scattered functionality Difficult to say. There are different repositories next to each other, but if you look at the NuGet packages, the relevant ones also editable directly from solutions. So some packages are both available as compiled packages and as source-code in the solution. In that sense there is scattered functionality. But mostly the shared functionality is centralized, so we think there is little scattered functionality.

Likelihood: Low. *Impact:* Low. **Relevancy:** Low.

#16 Risk of delocalized plans There is a shared document network drive, but the structure here really is chaotic. Besides that, this disk is terribly slow. So when design documents are created or other documents relevant for others, they are stored locally, emailed to relevant persons and sometimes put on the network disk.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

#17 Risk of iteratively changing reuse components In the FinCareClaim SaaS platform a number of core modules are used by the two main applications CarecoSoft and MedicoSoft. These modules are modified on a very regular basis. Hence, inside the platform the risk is already fairly high. On top of that, a large number of packages are shared among the different projects. Most of these shared packages are used by the FinCareClaim platform. updating such a package, from the viewpoint of whatever project, has a very high chance of impacting the FinCareClaim platform.

Likelihood: Very high. *Impact:* High. **Relevancy:** High.

#18 Risk of changes in product line assets at the product level Last-minute changes are common, but for crucial moments, for example at the offspring of a release, a separate branch is created to prevent other products to break down. So in that sense the risk of things going horribly wrong is low. However, during development changes at product assets with an ripple effect to product line assets are very common. The impact in terms of introducing bugs is manageable, but the impact on code quality and performance is higher. Overall, we think this is a risk of medium relevancy.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

#19 Risk of enhancement to a cross-cutting concern In terms of mutual change, there is a tight coupling between FinCareClaim core components and the application-specific components. Also, from the packages shared between applications, the FinCareClaim platform uses almost all of them. Hence, updating such a package from the perspective of FinCareClaim has a very high chance of having an impact on other applications. So, cross-cutting concerns can easily pop-up in a tightly coupled codebase, but currently the number of real cross-cutting concerns are low. The main cross-cutting concern is the workflow-manager and the logging functionality. If these are updated, the impact is very high, but the chances of this happening are not that high.

Likelihood: Medium. *Impact:* Very high. **Relevancy:** High.

#20 Risk of component granularity We did not conduct an analysis on the granularity level of the components, so the relevancy is unknown.

Likelihood: ?. *Impact:* ?. **Relevancy:** ?.

#21 Risk of circular dependencies We did not conduct an analysis on the circular dependencies of the components, so the relevancy of this risk is unknown. However, we learned that C#/.Net does not allow circular dependencies due to the way it handles compilation. So, it is safe to say that this is a low risk.

Likelihood: Very low. *Impact:* Very low. **Relevancy:** Very low.

#22 Risk of non-standardized configuration interfaces Deployment configuration is standardized in templates which are easy to use and is managed by visual studio. While there are always daily issues, there are not really problems here.

Likelihood: Very low. *Impact:* Very low. **Relevancy:** Very low.

#23 Risk of early binding of build-level dependencies NuGet packages are automatically build and updated in the solution via a Visual Studio plug-in. This is all managed by Visual Studio, so this risk has a very low relevancy.

Likelihood: Very low. *Impact:* Very low. **Relevancy:** Very low.

#24 Risk of making a composition by hand Builds are done automatically by Visual Studio for development or Jenkins for testing. Still, a release at a customer site can involve custom work like transferring files manually to the correct places. This however has nothing to do with the shared codebase, hence this risk is not relevant.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Very low.

#25 Risk of making an application/component reusable Currently at FinCare there is little effort being put in making components reusable, so it is not really a relevant risk. Rather maybe a missed opportunity.

Likelihood: Low. *Impact:* High. **Relevancy:** Medium.

#26 Risk of heterogeneous communication Between teams there is little communication and definition of any process regarding modifying shared packages, so at first sight this would seem a highly relevant risk. However, since the work is comprehensive and the guys involved are skilled, there is no real formal process required. The impact increases with the size of the codebase which increases with the number of products.

Likelihood: Medium. *Impact:* Medium (now). Increases with the size of the shared codebase. **Relevancy:** Medium (now).

#27 Risk of centralization in group based collaboration networks At FinCare updates on packages are all done by the senior developer. He can be seen as the gatekeeper. This means that when there are problems, he suddenly can become very busy. There is little formal documentation or planning as to what packages are going to get updated. Sometimes there is email communication about an update, a shout around or he walks around, but what normally happens is that an update is just pushed and any problems fixed on the spot. The risk is therefore highly relevant since there is a fair chance that the work, the amount of packages to manage becomes incomprehensible.

Likelihood: High. *Impact:* High. **Relevancy:** High.

#28 Risk of reuse experience level Because of the centralization of responsibility of the shared code, there is little knowledge sharing on the packages part. So other people without any experience, get very little chance to gain experience outside their project scope. Also, between the teams there is no reuse opportunity seeking. Teams work on their own products and while they work in the same implementation language, people don't come together cross-teams to see hey, can we reuse this and that in our application. In irregular intervals (say 1-2 months) there is a meeting with all developers where usually some general topic is taken to discuss with everyone. This kind of meetings are opportunities to share knowledge.

Likelihood: Medium. *Impact:* Medium. **Relevancy:** Medium.

§ 5.4 CONCLUSION

In table 5.6 we listed and sorted the risks according to relevancy. There are 6 risks with a high relevancy and two with a medium relevancy which may become more relevant in the future. We take these 8 risks as the shortlist:

- Risk of iteratively changing reuse components
- Risk of enhancement to a cross-cutting concern
- Risk of time-to-market pressure

- Risk of business value thinking
- Risk of operating in a volatile market
- Risk of centralization in group-based collaboration networks
- Risk of business philosophy focusing on short-term goals
- Risk of heterogeneous communication

This shortlist answers our third question of this study: *What are the relevant risks for FinCare?* In the next chapter we will relate the shortlist to the mitigation approaches found in the interviews to find suitable mitigation approaches for FinCare.

#	Risk	L	I	R
17	Risk of iteratively changing reuse components	VH	H	H
19	Risk of enhancement to a cross-cutting concern	M	VH	H
3	Risk of time-to-market pressure	H	H	H
7	Risk of business value thinking	H	H	H
1	Risk of operating in a volatile market	VH	M	H
27	Risk of centralization in group based collaboration networks	H	H	H
6	Risk of business philosophy focusing on short-term goals	VH	M ↑	M ↑
26	Risk of heterogeneous communication	M	M ↑	M ↑
5	Risk of political aspects	H	M	M
2	Risk of stakeholder influence	H	L	M
13	Risk of goal ambiguity	M	M	M
14	Risk of scope widening	M	M	M
16	Risk of delocalized plans	M	M	M
18	Risk of changes in product line assets at the product level	M	M	M
28	Risk of reuse experience level	M	M	M
25	Risk of making an application/component reusable	L	H	M
10	Risk of reusing immature components	M	M	M
8	Risk of prioritizing on mainstream product	L	L	L ↑
4	Risk of evolving standards	VL	M	L
11	Risk of unclear requirements	L	L	L
12	Risk of different interpretations of artifacts	L	L	L
15	Risk of scattered functionality	L	L	L
9	Risk of changing the business strategy	VL	M	L
24	Risk of making a composition by hand	M	M	VL
22	Risk of non-standardized configuration interfaces	VL	VL	VL
23	Risk of early binding of build-level dependencies	VL	VL	VL
21	Risk of circular dependencies	VL	VL	VL
20	Risk of component granularity	?	?	?

Table 5.6: Risks according to Likelihood (L), Impact (I) and Relevancy (R)

- Chapter 6 -

Finding suitable mitigation approaches

§ 6.1 INTRODUCTION

Up until now we have identified possible risks associated with requirements changes in a shared codebase environment (chapter 3), we have investigated how different units at Topicus work on a shared codebase (chapter 4) and from the risks we have made a shortlist of the relevant ones for FinCare (chapter 5). In this chapter we will answer question 4 of this study: *What risk mitigation approaches are suitable to implement by FinCare to mitigate the potential risks?* We will do this in three steps:

1. Map solutions and best practices from interviews to the shortlist.
2. In-depth discuss best practices and solutions.
3. Determine suitability of approaches for FinCare

§ 6.2 MAPPING SOLUTIONS/BEST PRACTICES TO SHORTLIST

From the interviews we have found a number of best practices of working on a shared codebase and dealing with changing requirements. In this section we try the map these best practices to the risks from the shortlist.

Risk of iteratively changing reuse components (#17) From the interviews we learned that the following solutions can mitigate this risk:

- In the case of sharing components with other business units, give other units only read rights and no commit rights.
- When sharing code across units, work on your own branch.
- Prevent doing small patches in a production environment.
- Have clarity about responsibilities of components.
- Have more knowledge sharing sessions about both the available components and the latest developments for all applications.

This risk is about changing a component shared among applications in consequent iterations. What can go wrong here is that one project team can directly affect the codebase of other project teams. Also, iteratively changing means that the component is in active development, volatile, hence its functionality may be unstable.

Dealing with this risk seems to ask for clear responsibilities, code ownership and knowledge sharing about the contents and activities with respect to the codebase. There are two assumptions related to this. The first is that if someone is responsible for a component, modifications are supervised by this person and any contingencies are managed centrally. The second assumption is if you know enough about all components, you automatically have the required insight to foresee the impact of changes, hence you can prevent negatively affecting other's code.

We think that for teams working according to agile principles like the interviewed units, iteratively changing components is common practice and therefore implicitly a very highly relevant risk. So the real problem in this situation is not the iteratively changing itself, but rather that updates from one day to the other can ripple to another project's codebase. To prevent this, versioning your components and consciously releasing them in versions seems a better and more direct solution. Interviewees at unit A indicated actively thinking about doing this and unit B already does so.

Risk of enhancement to a cross-cutting concern (#19): From the interviews we learned that the following solutions can mitigate this risk:

- In the case of sharing components with other business units, give only read access and no commit rights.
- In the case of sharing components with other units, it is preferred to work physically in at least the same building.
- Use selenium tests to guarantee front-end stability and unit tests for core components and core web-services to guarantee core component stability.
- Make automatic nightly snapshot builds on the development environment with automatic execution of tests.
- Have more information about upcoming updates on external components.
- Implicitly test shared components using regression tests.
- Have a core component team responsible for development and maintenance of the shared codebase.
- Have more information in advance about upcoming changes.
- Have more communication about updates on the shared codebase.
- Have more unit tests and more knowledge as to how to structure and approach testing.

The most direct problem associated with this risk is the same for the previous risk. When a cross-cutting concern is updated, it has a very high chance to ripple through the codebase to other applications if the cross-cutting concern is implemented in a shared component.

Since a cross-cutting concern is more a technical problem than the previous risk, the solution of relying on automatic and implicit testing is logical. Knowing about upcoming updates or having someone responsible for a component does not directly help to mitigate this risk, but it supports the development process which indirectly helps mitigating this risk.

Risk of time-to-market pressure (#3) From the interviews we did not find any direct best practices regarding this risk.

The first problems related to this risk is an increase of technical debt. Technical debt is a metaphor used to describe the incremental postponed work for short-term gain in agile development environments (Brown et al., 2010). The second problem underlying this risk is employing a platform in production while its components are still immature and a decrease of re-usability of platform components.

Risk of business value thinking (#7) From the interviews we learned that the following solutions can mitigate this risk:

- Have a vision on asset reuse.
- Get support from top management for resource allocation for reuse activities.

The problem with business value thinking are the same as with the previous risk, but have a different cause. Business value thinking has directly to do with the culture and management support for resources spent on platform development. When there is no support from top management to spent time on platform development (making components reusable, refactoring, updating), project teams will simply not spent time on it if there it is not in the scope of meeting their own project goals.

Risk of operating in a volatile market (#1) The main problem associated with this risk is having to adapt your products because of changes in the market. From the interviewees no concrete solutions for this problem were discussed. However, as indicated before, for FinCare the aspects of this risk which have an impact on the codebase are dealt with using the workflow manager (process configuration). So while the risk is highly relevant, a good solution is already available for FinCare. We therefore exclude this risk.

Risk of centralization in group based collaboration networks (#27) From the interviews we learned that the following solutions can mitigate this risk:

- Everyone can edit COBRA components.
- COBRA is of, by and for everyone, creating a social self-regulating system.
- Have a contact person for certain components.
- Have a core component team responsible for development and maintenance of the shared codebase.

For this risk the associated problem is having a bottleneck in the development process on the shared codebase. The solutions all suggest to spread the responsibility of the shared components. For example to allow editing on the shared codebase by everyone, relying on a social self-regulating system to assure code quality. The problem with this approach is that it is difficult to know who is working on what component and what components are used where. This is tackled by the suggestion of some interviewees to have a contact person for a number of components. We think that making this person responsible for the component is not a good idea, rather he should be the expert for some components, to whom people can go to for help or information.

Risk of business philosophy focusing on short-term goals(#6) From the interviews we learned that the following solutions can mitigate this risk:

- The positive side of having a actively developed shared codebase are always overshadowed by the the negative aspects or experiences. Having more awareness of why there is a codebase stimulates reuse, design for reuse and common ground for resource allocation.
- Support from top management for resource allocation for platform development.

The problems with business value thinking are the same as risk 3 and 7. But, as with risk 7 the cause is slightly different. A business philosophy is more abstract than time-to-market pressure and business value thinking and has more to do with the prevailing development culture, the ‘mind-set’ of the teams. Support from top management then is a logical and important

solution, but if there is no culture of ‘having and investing in reusable assets is something we value highly’, then there still will be a lot of technical debt, usage of immature components and lack of reuseability. Multiple interviews mention that the negative stories of reuse, the obvious stigmas of ‘what’s in it for us’, ‘costs a lot of time’, ‘no direct visible value’ overshadow the benefits, which for products with a SaaS delivery model are very large.

Risk of heterogeneous communication(#26) From the interviews we learned that the following solutions can mitigate this risk:

- Have a dedicated person for shared components.
- Separate development, support and implementation in different organizational parties to enforce separate responsibilities.
- Give everyone write permissions to to the component to support a social self- regulating system.
- Have a gatekeeper for codebase modifications.
- Send emails when updating some component.
- Have self-responsibility for communicating changes and writing of unit tests.
- Have more awareness that when large modifications are done, this can have an impact besides your own project scope.
- Have more communication when someone is changing a shared component.
- Be physically located close to other users of your code.
- Have a contact person for important shared components.
- Have a core component team.
- Have a component leader.

The main problem behind this risk is that there is no communication about updates resulting in less time and thus opportunity for developers to anticipate for the change. The solutions scream all the same: either have no process and rely on the awareness and professionalism of your developers to inform people when updating components or have a clear policy of what to communicate to whom when updating some shared component, depending on who is responsible for components. The notion of wanting a component leader or some core component team sounds more like a lack of process than a direct solution.

§ 6.3 DISCUSSION OF MITIGATION APPROACHES

From the shortlist of risks and the mapped mitigation approaches we notice that a requirement change on a shared codebase encompasses 3 solution domains: dealing respectively with technical implications, organizational culture and communication and collaboration issues. In this section for each of these solution domains we will:

- Give a causal diagram to show the relation between solutions, problems and risks.
- Discuss to what extend the solutions can help in solving the problems, supported by results from a questionnaire conducted at FinCare.
- State recommendations both for FinCare and Topicus.

6.3.1 Questionnaire

In the next sections causal diagrams will be used to show the relation between solutions, problems and risks. For each problem and solution we have defined one or more statements which we asked

everyone at FinCare to evaluate according to a 5 point Likert scale. For each statement we will show the Net Promoter Score (NPS) which is the net percentage of positive (totally agree and agree) minus the percentage of negative (totally disagree and disagree) scores. The detailed results for each statement can be found in Appendix C.

6.3.2 Dealing with technical implications

The technical implications are the every day issues when updating code from a shared codebase. For FinCare two risks are associated with this: the risk of iteratively changing reuse components and the risk of enhancements to a cross-cutting concern. Three problems are related to these risks: lack of stability of component functionality, unintentionally rippling of changes to other applications and a lack of knowledge about the impact of a change. In Table 6.1 we see a lot of support for the associated statements, indicating that this already currently are issues for FinCare.

The direct solutions from the interviews for this are versioning of components and writing unit tests for core platform components and web-services. In section 2.3.4 we also gave an overview of different impact analysis methods, which indirectly also helps dealing with the associated problems. The relation between solutions, problems and risks is given in a causal relation diagram depicted in Figure 6.1.

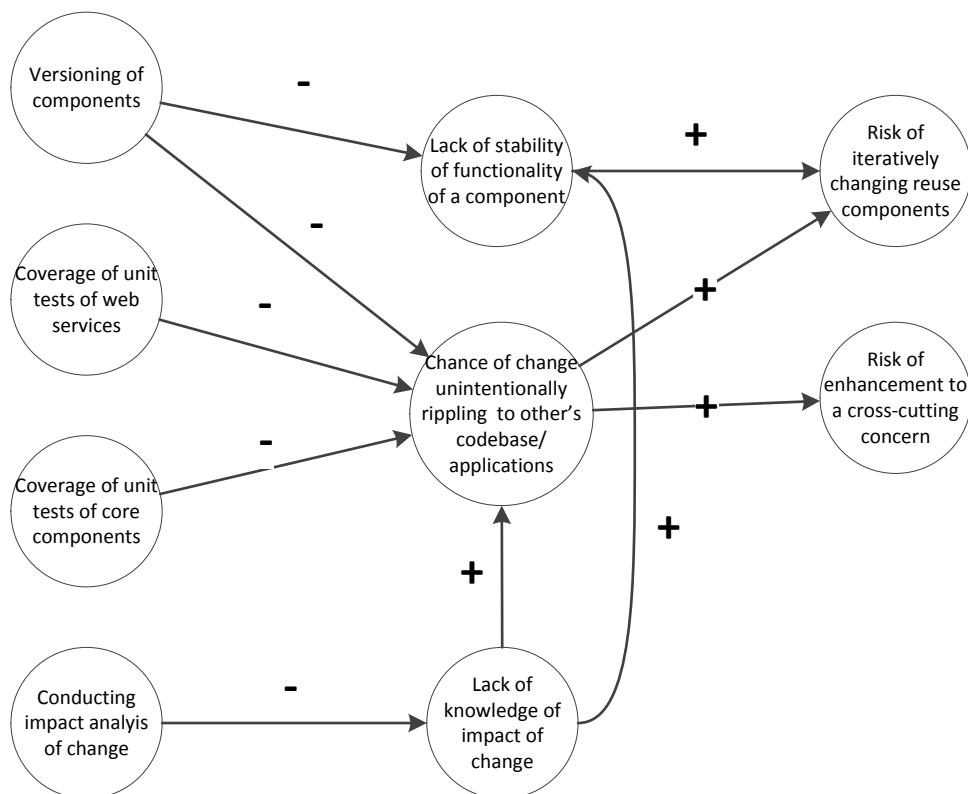


Figure 6.1: Causal relation diagram of dealing with technical implications

6.3.2.1 Versioning

Versioning components yields two things: 1) it ensures that changes do not directly ripple to other applications and 2) it enforces to plan ahead for modifications on components. Both

Problem	Statement	NPS
<i>Dealing with technical implications</i>		
Chance of change unintentionally rippling to other's codebase	Components which I use are often updated by others outside my project team.	60,00%
Conducting impact analysis of change	When I push a modification to a component I'm certain that no defect is introduced in another application.	-57,14%
Lack of knowledge of impact of a change	When I push a modification to a component I'm certain of the impact for other applications.	-42,86%

Table 6.1: *NPS of problems dealing with technical implications*

make sure that components in some version are stable; they will only change when their version is updated. Currently versions are given to NuGet packages, but just giving a component a version number is not enough to assure code stability. As indicated by (Stuckenholtz, 2005), version numbering alone are not expressive enough. Only when a consistent component release model is used stating when a change should trigger a new version, versioning contributes to component stability.

6.3.2.2 Testing

In all interviewed units, it is the responsibility of the individual developer to write unit tests when they think it is necessary. For the interviewed units there doesn't seem to be a testing policy. The result is that interviewees found it difficult to estimate how much test coverage there is. Also, multiple interviewees indicated that the test coverage is low and more testing is always better. However, making the trade-off between writing extensive unit tests and finishing the bugs and features for some upcoming release mostly results in the latter.

Having complete code coverage does not seem to be a requirement, most developers are smart and pragmatic enough to tackle bugs and problems fast enough. However a desire seems to be to have at least automated unit tests for the core components for a platform. What units do now is rely on regression testing (following a test script to test a list of test cases), which then indirectly invokes the components.

6.3.2.3 Impact analysis

Visual Studio allows for dependency analysis, but only from the active solution. It never enables you to show the complete picture. From all the methods above, at FinCare only call graphs and to some degree dependency analysis is utilized, but as said only from the perspective of the active solution. With the diagrams from chapter 5.2 we were able to get much more insight in the internal structure from the code than from all tools available for Mercurial and Visual Studio. Techniques like execution traces, program slicing and probabilistic models are not relevant for FinCare and Topicus, but history mining and architectural models are. Visualizing architectural changes and decompositions can give fast insight in the impact of requirements changes on a shared codebase. For Eclipse and Java experimental tools like Ariadne exist which focus on visualizing socio-technical dependencies (de Souza et al., 2007).

Solution	Statement	NPS
<i>Dealing with technical implications</i>		
Component versioning	Versioning components is a good approach to assure component stability.	70,59%
Coverage of unit tests of web services	Striving for complete code coverage of shared web services is a good approach to prevent unintended effects in web services in other applications.	37,50%
Coverage of unit tests of core components	Striving for complete code coverage of unit tests for core components is a good approach to assure stability.	37,50%
Conducting impact analysis of change	When modifying a component I'd like to know better what applications use this component.	58,82%

Table 6.2: *NPS of solutions dealing with technical implications*

6.3.2.4 Producer-consumer

Within the Topicus holding, small groups of units may collaborate on shared components. In theory all units should be able to share components with each other, but since 1) units use different implementation languages, 2) units use different repository software (central version control like Mercurial, SVN or GIT) and 3) units do not standard have read-access to each other's repositories, there is little sharing of functionality in general.

Also, the incentives for sharing with everyone are low. Units operate in different domains, have different customers, units are separate companies working rather isolated from each other, so why should they? Topicus is a lean company, overhead is a black term. Having to make agreements about working on code together smells like overhead, so we don't want it; that is the overall tenancy. One unit, unit A, has solved this by saying: hey you can use our components, but we give no guarantees, it is your responsibility if you use them. But since they not actively communicate about changes, or advertise the nice components they have built, only one unit is using some of their components.

Unit B on the other hand actively works together with unit C and with FinCare on some functionality. Between unit B and C there is merging of code in both directions. There are some occasional frustrations, but overall it goes well. So, why does this work? Because unit B and C visibly have an overlap in functionality, they share domain-specific assets in which both have a stake. However, the components they share are generic for their application domain only. For generic components suitable for all domains, without any relation between two units, the other units would 1) either not know about such component or 2) don't want to hassle of coordination between the units and build it themselves.

We argue that units should be able to publish and advertise domain-generic and overall-generic components in an easy way for other units to reuse. Units should be stimulated to version components and 'release' them. Consumers of the components should be (automatically) notified and make their own judgment as to pull the update or not. The responsibility of using such components lays with the consumer, the producer only should develop with the intention to keep components as generic as possible, provide documentation and advertise the goals and purpose of the component. This involves little overhead, but a lot of reuse opportunity. Currently, there is no such mind-set in Topicus. A pity, since it would bring together a lot of knowledge and stimulate innovation. For more on reuse models, see (Fichman and Kemerer, 2001) and (Bosch, 2001).

6.3.2.5 Recommendations for FinCare

Firstly, we think that releasing components in versions is a common practice which can greatly reduce any ripple effects for the FinCareClaim platform and enforces a slightly more structured approach in developing a software product line. In Table 6.2 a high NPS (70,59%) is given for the associated statement, so there is already agreement that component versioning is a good idea.

Secondly, we stated that having unit tests for all core components with a high level of coverage is a good idea since it reduces the risk of unintentionally breaking something down in another application. In that respect, the results from the questionnaire are less positive (NPS of 37,50%). A few respondents commented that having high levels of unit tests is always better, but not always feasible or really necessary. So this should be a situational judgement. The same response was received with regards to unit testing web services. Currently FinCare has no sharing of components with other units, so the solutions related to that are not relevant for FinCare, but rather for Topicus as a whole. In short:

- Incorporate component versioning for the core platform components
- Strive for a high coverage level of automatic unit tests on the core platform components

6.3.2.6 Recommendations for Topicus

- Stimulate sharing of components by incorporating one type of version control for all business units in the holding, for example mercurial or GIT.
- Stimulate sharing of components by giving all units read-access of all repositories.
- Have units ‘deliver’ generic, sharable components in versions. The policy should be simple: others can use components, updates are communicated to all consumers and units should intent always to make components as generic as possible, but the responsibility is with the consumer. Consumers must be able to fork their own version at all times.

6.3.3 Dealing with organizational culture

Three risks are related to organizational culture and have very much to do with each other: risk of time-to-market pressure, risk of business value thinking and the risk of business philosophy of focusing on short-term goals. The reason they are related is because they all have the same underlying problems: the usage of immature platform in production, lack of reusability of components and technical debt.

From the results of the questionnaire we see that there is disagreement about the immaturely using a platform in production: 6 agree, 4 neutral and 6 disagree giving a NPS of 0,00%. There is medium agreement about the reusability of the components (NPS 50,00%) and low agreement about the existence of technical debt (NPS 23,53%). This supports our statement that currently the problems are comprehensible, but from our analysis of the codebase we found that due to the high level of coupling of the FinCareClaim applications, the relevancy will increase with the size of the platform and the number of supported applications.

From the interviews we found 3 solutions for these problems: propagation of a positive attitude towards reusable assets, support from top management for resource allocation and a fair and transparent billing of platform activities. The relation between solutions, problems and risks is given in a causal relation diagram depicted in Figure 6.2.

6.3.3.1 Propagation of positive attitude towards reusable assets

Cost transparency For work on a shared codebase it is difficult to be both transparent about costs to your customers and fairly distribute costs among teams. We see two kind of cost

Problem	Statement	NPS
<i>Dealing with organizational culture</i>		
Usage of immature platform in production	At FinCare a platform is put into production too soon, with too many open issues.	0,00%
Lack of reusability	Components at FinCare are reusable and directly employable in other projects.	50,00%
Technical debt	At FinCare refactoring and code cleaning is pushed forward or postponed a lot.	23,53%

Table 6.3: NPS of problems dealing with organizational culture

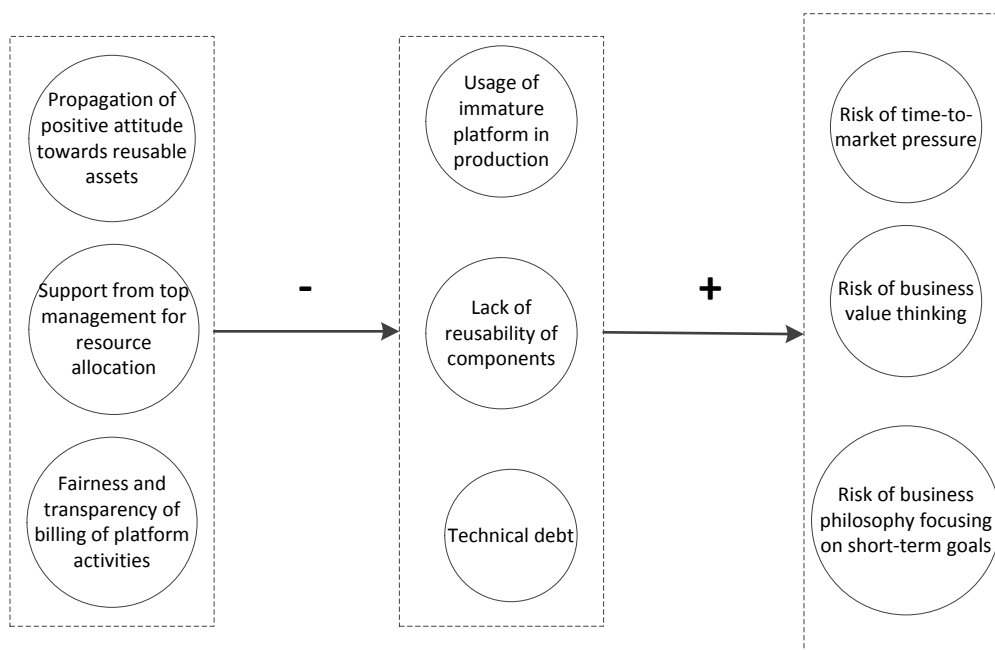


Figure 6.2: Causal relation diagram of dealing with organizational culture

transparency problems.

The first is *cost transparency for customers*. In the case of a software platform with multiple products each with their own group of tenants, how do you bill costs on a component shared among the products? If you are very transparent on the billable hours, customers can argue that they don't want to pay for work effort which is not directly related to the features and service activities they signed for. Customers don't care about the very generic setup of your codebase which makes it easier to roll-out new products. In order to make sure that there is a clear budget for platform work, regardless of who (some customer or developer) initiated the work, there must be a dedicated budget for this payed via the license fees. To customers, it must be made clear that this is just what makes a SaaS solution cheaper in the long run.

The second transparency problem is *transparency of cost sharing among project teams*. What happens in practice is that work effort on a component is initiated from a particular project, because they need some modifications for their product. From that point, all work effort which also is of benefit for other project teams is done by the initiating team. This means that they ought to pay for the work effort as well. It comes right from their budget, hence it impacts their project performance. If there is no equal give-and-take among teams of this kind of work effort on the shared codebase, the cost distribution can becomes unfair. Teams are less willing

to invest actively in generic code for sharing if other teams don't do it. Business units must have a clear hour billing policy for internal investments in platform code.

Reuse opportunities New functionality is constantly written and it is very doubtful that every method written is completely unique in what functionality it offers. Duplicated functionality is a fact, even more in a situation where business units work individually next to each other. Spotting reuse opportunities therefore could be a very resourceful activity, both in terms of knowledge and time, both inside a business unit and between business units. From interviews we learned that in a unit having regular sessions with developers from different teams already stimulates this, as long as it is a recurring item. In unit A everyone can contribute to the shared codebase, but since there is little knowledge sharing about what is already available, generic made components stay anonymous and the number of contributions grows larger and larger, of which the larger part is never reused.

Opportunity for reuse has two categories: reuse of application modules (e.g. a user management module) and reuse of generic components (e.g. a automatic form generator from a database structure). For the first, the opportunity is easier to spot and often a better case can be made to allocate resources. Also, advertising of such components is not required since they are only used in some project or product-family. For the second category, a case is harder to make because of the refactoring reasons stated earlier. Also, these components are easier forgotten if not actively used by other products.

We think that the everyone-can-contribute attitude of unit A is good for the second category of reuse, with the addition that newly added components are discussed in plenary sessions to get feedback from others and advertize its existence. For the first category, the everyone-can-contribute attitude is not important, rather you must look at your domain and try to spot generic opportunities in the desires and feedback customers provide. This is something unit B currently actively does, unit A has this outsourced to its implementation partner.

6.3.3.2 Support from top management

From the interviews we learned that time spent on refactoring is found to be 'too little'. We argue that this is because of self-preserving reasons. Refactoring is only initiated when it directly benefits the product someone is working on. Otherwise, because of the short release cycles and tight planning (or lean development), there is always something directly beneficial for the product to do. Moreover, refactoring of *shared* components implies that you are taking a risk affecting and breaking down other applications outside the scope of your own project and spending project resources on other projects. In a culture where costs are transparent to customers and time-to-market is fast, this discourages refactoring. Who pays for reuse effort? Support from top management is needed allow resource allocation for platform development.

6.3.3.3 Recommendations for FinCare

For FinCare 'top management support', or 'cost-transparency' are not direct usable solutions. Cost transparency is not always feasible when you are happy enough to get some customer to sign a new contract for a few months, which is the case for FinCare often enough. Their products are not in the life-cycle phase like the products of unit A and B where cost transparency is easier to achieve because of the supplier bargaining power of the units.

In the questionnaire (see Table 6.4) when asked about hour billing respondents did not indicate really any problem here. Rather we would say this is something to consider in the near future when products of FinCare have a more stable position in the market.

Solution	Statement	NPS
<i>Dealing with organizational culture</i>		
Propagation of positive attitude towards reusable assets	I would like to see a more positive attitude with respect to reusable components.	5,88%
Support from top management for resource allocation	Enough time is available for personal initiatives to improve components.	6,25%
Fairness and transparency of billing platform activities	I can honestly justify hours spent on components.	54,55%
	Towards customers hours spent on components can be justified in good fashion.	42,86%

Table 6.4: *NPS of solutions dealing with organizational culture*

But we would say that there should be more collaboration between the teams. There is no active reuse opportunity seeking cross-projects and project teams know little about each other's road maps, while they certainly have an overlap in their codebase. While there is no real agreement about the desire for a more positive attitude with respect to reusable components (NPS 5,88%), there also is low agreement with respect to enough room for initiatives (NPS 6,25%). We therefore suggest the following:

- Try to, in collaboration with each other, regularly and actively find reuse opportunities spanning the different applications. For example, by making it an agenda point for plenary sessions or developer meetings. Also, share newly created components with each other. Present them, discuss them. Make sure resources are made available to allow development of such components.
- Allow more time and freedom for refactoring and uncoupling of FinCareClaim-components. Currently they are tightly coupled, actively under change and form a risk since they are the core functionality.
- Involve more people in the development and maintenance of the NuGet packages
- Propagate the planning and road maps of products more actively to everyone in the unit. Since the scope may be medium in size, it is important to communicate it as fast as possible. Any contingencies on a higher, product design level can be anticipated on faster. Also, knowing what the other project teams are up to makes it easier to bring up ideas for the shared codebase.

6.3.3.4 Recommendations for Topicus

- Develop a Topicus-wide policy for billing non-project dedicated hours as part of a standardized SaaS delivery model. Stimulate sharing of experiences of hour logging/billing for refactoring or, non-project dedicated efforts. How to bill this, have a healthy business model and still be transparent to customers?

6.3.4 Dealing with communication and collaboration

Two risks are related with dealing with communication and collaboration issues: the risk of centralization in group-based collaboration networks and the risk of heterogeneous communication. The underlying issues here are the amount of bottleneck in the development and the amount of communication about codebase updates. The relations of the solution for these issues are depicted in a causal relation diagram in Figure 6.3. From the questionnaire we see that there is disagreement with respect to communication and collaboration. While the NPS values (see

Table 6.5) are rather neutral, this is because the number of agrees and disagrees counter each other (see Appendix C) rather well.

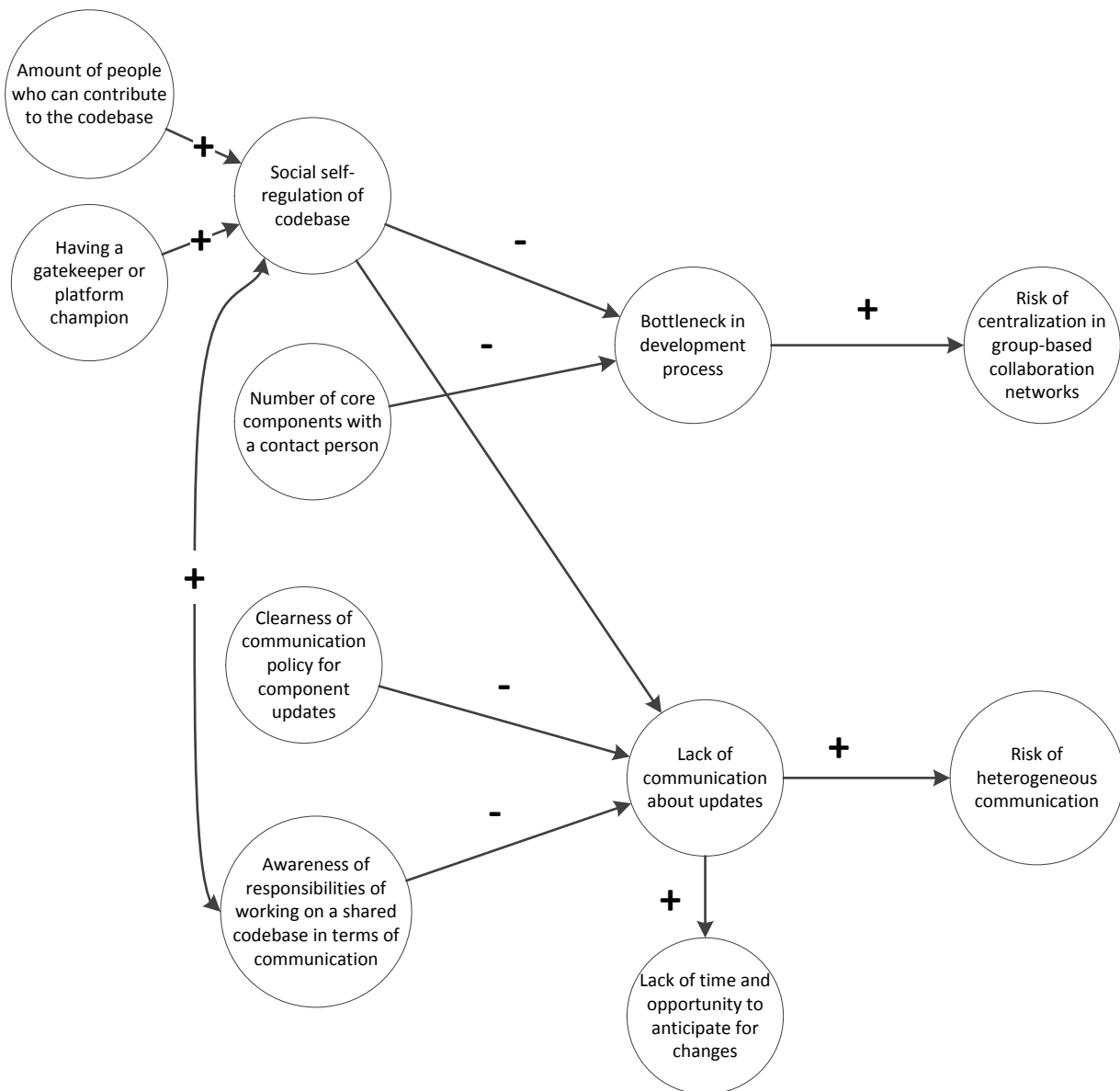


Figure 6.3: Causal relation diagram of dealing with communication and collaboration issues

6.3.4.1 Social self-regulation of codebase

Social self-regulation of the codebase means that you rely on the professionalism and skills of developers to assess the impact of some modification and handle accordingly. Also, it is considered professional to spot reuse opportunities and take initiative to make the team or other teams aware. This only works in an environment where everyone has write access on the shared components. All this requires communication guidelines (what to communicate when to whom) and awareness of responsibilities. Also, a central person of authority, a reuse champion or a codebase gatekeeper should have the role of expert, checking for quality. Such a person should not have end-responsibility for the codebase, that should always be with the team or developer

Problem	Statement	NPS
<i>Dealing with communication and collaboration</i>		
Bottleneck in development process	Modifications on components are always done by the same person.	-12,50%
	I know who to notify when I modify a component.	-7,69%
	I always notify the correct people when modifying a component.	-8,33%
Lack of time and opportunity to anticipate to changes	When a component is modified, people inform each other well.	-25,00%
	When a component is modified I have enough time in advance to deal with the potential impact of the change.	31,25%

Table 6.5: *NPS of problems dealing with communication and collaboration*

conducting the modification. This is a rather vague solution, but we will try to explain below by discussing the mentioned aspects.

6.3.4.2 Contact person for components

If there is a central person of authority sitting on the shared codebase, units must be careful. A central, senior person can be a champion in the sense that other developers follow his example and come to him for advice and help. However, if this single person alone is responsible for the design and implementation of the reusable assets, then there is the risk of a bottleneck in the development process. A single person dictating the interests of all applications is an unhealthy situation. Rather, an advisory role and stimulating reuse from a code quality perspective is favored.

6.3.4.3 Communication

When a requirement changes, a component is updated, a web-service is altered, other applications and people may depend on it and thus are affected. Informing them then is not only a courtesy, but also may be vital to the business if some application becomes defect because of the change. The question then is, to whom to communicate the change, when and what to say. Not everyone may be interested in the change, you might not even know who is interested. Even if you would know everyone in the organization who would be interested in knowing about the change, what to communicate? Can they respond, do they have a saying in the change? When to communicate, before the change? After?

Also, from the interviews we learned that mental and physical barriers prevent good lines of communication. Having to walk to ‘the guy in the next room’ is a large enough barrier not to communicate. Having to walk 5 minutes to the next building to check something before changing something is a large enough barrier not to communicate, or inefficiently spent time in composing and responding to email.

Regardless of the nature of the change, three important elements seem coming back: knowing who to communicate to, by what means and what to communicate. The first question can be answered by 1) knowing what application and assets are affected by the change and 2) knowing who works with/on these assets. The second question is a matter of personal preference or unit culture or can be, combined with question 3, part of a unit’s policy of communicating change.

Not only when actually making the change communication is desired, but also in advance. Knowing when updates can be expected so that teams can prepare. Also, communication of

plannings allow teams to probe for the change, asking around about it. On a more higher level, communicating road-maps of applications allows teams to design for reuse.

6.3.4.4 Knowledge sharing

From the interviews we learned that knowledge sharing is a topic which is central to the issues regarding a shared codebase. Having a good platform to share knowledge then is a requirement. Questions here are, what platforms are suitable to incorporate? What knowledge do we want to share? Who is responsible to provide what information? This is a broad domain outside the scope of this project. We only acknowledge the importance of having a company-wide policy to stimulate knowledge sharing by making resources available to facilitate this. For Topicus a centralized knowledge sharing solution may not be optimal, because different groups of units with an interest in each other may have different knowledge sharing desires. Rather groups of units must be able to use their own knowledge sharing platform. This must be stimulate from the top and experiences shared with other units. Important knowledge sharing for knowledge intensive industry like software development is the location of expertise (Faraj and Sproull, 2000). Interviewees indicated that knowing who is using what component is an important, but often difficult to answer question.

6.3.4.5 Project structures

Units at Topicus are mostly organized in product-driven development teams. Teams are dedicated for a long time on a specific product or set of products. There are also teams working in a more project-driven fashion and for example at Unit D a combination of component-driven and project-driven development is used.

We think that feature-driven development has a very high potential for the more larger units of Topicus, like unit B. The issues with working on a shared codebase there are communication or responsibility issues. The first thought than is to organize teams around modules or components, like component-driven development. We see that this can result in team lock-in and organizational overhead, something which may be fine for a unit like Unit D but in general is not something which fits the culture Topicus embraces. With feature-driven development, knowledge sharing, communication lines and responsibilities are implicitly defined.

The situation described above is typical for the units A, B and FinCare and can be labeled as a product-driven development structure. Unit B considers to move towards a component-driven structure, since it allows for clear responsibilities and resource allocation. We argue that this is a bad idea, since the scope of the modules are too small for this and it implies a level of coordination which does not fit in the culture of Topicus.

Rather, we would say that the feature-driven development structure is better and far more suitable for unit B. the most important attribute of feature-driven development is that knowledge sharing becomes implicit, since people work on different applications over time. We think that such a structure only works when a unit is medium sized, so for FinCare this would not work.

6.3.4.6 Recommendations for FinCare

Firstly, in chapter 5 the risk of heterogeneous communication was given a medium relevancy, with the potential to increase in the future. The application portfolio and the codebase are comprehensible at the moment, so a contact person for components or a strict communication policy are not directly needed. Still, respondents from the questionnaire that having contact persons is a really good idea (NPS 70,59%) and also a simple but clear communication policy would be an improvement (NPS 82,35%).

Solution	Statement	NPS
<i>Dealing with communication and collaboration</i>		
Social self-regulation of code-base	Everyone should be able to modify components as their own responsibility.	-52,94%
Contact person for core components	A contact person for each component is required in order to have clear lines of communication.	70,59%
Communication policy	A simple but clear communication policy for component updates would be an improvement.	82,35 %
Awareness of responsibility	There is enough awareness about the responsibilities of working with components.	50,00%

Table 6.6: *NPS of solutions dealing with communication and collaboration*

Secondly, in the questionnaire there is agreement about giving everyone write-access and be self-responsible when editing components (NPS -52,94%), indicating that this is not a favored approach. Yet, respondent did agree that there is enough awareness of responsibility (NPS 50,00%).

We argue that, (1) since there is a medium risk for having a bottleneck in the development process and (2) there is lack of good communication when updating components, just involving more people in the development of the shared codebase contributes to solving both problems. To assure code quality, the role of a central authority must be preserved as quality controller, as an expert and not as the single person responsible for the codebase. In short:

- Have a simple but clear communication policy when updating packages. For example, when updating a component, small or large, communicate it to all developers, analysts and team-leaders a few days in advance.
- Involve more people in the development of the shared codebase.
- When growing larger as a unit, avoid strong separation and lock-in of teams. Implement aspects of the feature-driven development paradigm to keep knowledge spreading. Full-size feature teams might, due to the size of the current projects teams be feasible. But why not rotate developers and analysts once in a while?

6.3.4.7 Recommendations for Topicus

- Stimulate sharing of experiences of knowledge sharing between units. A discussion about how to facilitate knowledge sharing is going on at a lot of units, but a centralized solution may not be the silver bullet. Knowledge sharing needs to be tailored for the needs of groups of units who have an interest in each other. So as a start a small group of units need to start working together, for example in the Care-domain. Central here are knowledge questions like: who is responsible for some component, who is using some component, who is working on some component, what components are available for reuse, where is the documentation of some component located, what is the planning road map for some product, etc.
- Stimulate active development of shared components, but by different people by implementing elements of the feature-driven development paradigm. Avoid component-driven development. While the responsibilities may be clearer, it does not stimulate active refactoring, fresh ideas about shared code and in general the knowledge of what is located in the shared codebase. Teams and people can become locked-in. As long as there are good knowledge sharing facilities, the risks associated with updating cross-cutting concerns can be mitigated.

- Avoid centralization in the development cycle of shared components. Authority persons should have an advising and quality checking role, but should not have responsibility of the design of the shared functionality. That lays with the project team.
- Have a simple but clear unit-wide policy as what to communicate and to whom when updating shared components
- Invest in facilities to enable fast finding out what applications use what components
- Groups of units with close collaboration on the same code need to be located at least physically in the same building

§ 6.4 CONCLUSION

In this chapter we have mapped the best practices/solutions from the interviews to the shortlist of most relevant risks for FinCare. While discussing this mapping we found that the relation between the solutions and risks can be grouped into three solution domains (dealing with technical implications, dealing with organizational culture and dealing with communication and collaboration), each with their own causal relations. We discussed the suitability of the solutions according to these solution domains in the context FinCare and stated our recommendations as well as general recommendations for Topicus.

- Chapter 7 -

Conclusion

§ 7.1 ANSWERING THE PROBLEM STATEMENT

The problem statement for this study is: *How can FinCare mitigate risks associated with requirement changes in an agile development environment where multiple projects share functionality located in a shared codebase?*. In chapter 2.4 we define 4 research questions to answer this problem statement. The first research question is: *What are the risks of a shared codebase environment with respect to changing requirements?*

We answered this question by conducting a literature study of industrial case study in the period of 2000 and 2012 to find recent issues from practice. The result is a list of 28 risks covering a very broad range of topics on a organizational, process and technical level.

In the second research question we look for techniques, best practices, approaches to deal with these risks: *What mitigation approaches can be used to mitigate these risks?*. We answered this question by conducting 8 interviews in 3 different business units at Topicus. In these interviews we identified issues, solutions and desires the interviewees have with respect to working on a shared codebase and dealing with changes.

In order to answer the third research question (*What are the relevant risks for FinCare?*) we conducted an in-depth analysis of the codebase and the FinCare products and created a shortlist of the most relevant risks for FinCare. This resulted in a list of 7 risks:

- Risk of iteratively changing reuse components
- Risk of enhancement to a cross-cutting concern
- Risk of time-to-market pressure
- Risk of business value thinking
- Risk of centralization in group-based collaboration networks
- Risk of business philosophy focusing on short-term goals
- Risk of heterogeneous communication

The final research question is: *What risk mitigation approaches are suitable to implement by FinCare to mitigate the potential risks?*. To answer this research question we took the solutions we got from the interviews and related them to the shortlist. In section 6.3 we structure and elaborate on the suitability of these solutions. As the final result for this study and also for answering the actual problem statement, we structured the risks accordingly to 3 solution domains: dealing with technical implications, dealing with organizational culture and dealing with communication and collaboration.

Dealing with technical implications The risk of iteratively changing reuse components and enhancements to cross-cutting concerns both encompass problems dealing with the technical implications of some requirements change. The two main issues for FinCare are stability of functionality for components (the core components of the FinCareClaim platform for example) and the chance of some change unintentionally rippling to another codebase/application.

For the first issue component versioning seems a good best practice to use for FinCare. Versioning forces to plan ahead and is a self-protecting mechanism to prevent changes to directly ripple through other products, so it also addresses the second issue. Besides component versioning, making sure that all core components and shared web-services have a high level of unit test coverage also prevents for unintentional ripple effects. A high level of unit test coverage on a product level seems tedious for an organization like FinCare, but for core components this can help dealing with the impact of a modification on a shared component from a project's perspective on another project.

Besides these two solutions (component versioning and testing) we also think that having some direct method available to conduct bottom-up impact analysis helps mitigating these two risks. One of the big questions developers have is which applications use which components. To solve this, FinCare must have a tool available which shows the interdependency of the components between applications/projects spanning all repositories.

Dealing with organizational culture On a business and organizational level, three risks were found to be relevant for FinCare: risk of time-to-market pressure, risk of business value thinking and the risk of a business philosophy focusing on short-term goals. All risks can be related to the following problems: usage of immature platform in production, lack of reuseability of components and technical debt. We found three solution directions for these three problems.

The first is a requirement for the organization to actively propagate a positive attitude towards reusable assets. The reason this is important is that if there is no culture of 'having reusable assets is a good thing and we know that this takes an investment upfront with no clear direct results', there is a high chance it will never happen.

Secondly, to encourage this attitude top management needs to plan for resource allocation and must support work effort required to develop good platform assets. The third point relates to this. Because of the project-driven development approach, work effort on a shared codebase always comes for the larger part because some project initiates it because of their own local needs. Hence, they need to spend their project resources on it. If there are no incentives for a project team to invest in some feature to be available for other projects as well, they will only do the required work to meet their own project demands. So thirdly, to solve this, a fair and transparent hour billing system is required where teams can spend time on other projects.

The remaining question here is how to bill this to customers. We think that platform activities are both an investment and part of the reason why a SaaS solution is cheaper for customers than a one-off software solution. Depending on where in the life-cycle a platform is, the money should come from a different source. If the platform is immature and customers are few (like FinCareClaim SaaS at FinCare), platform effort is an investment for the business unit. For the products from unit B from the interviews, platform effort is billed via their licenses. Their customer-base is more stable, larger and thus unit B can afford to project their own product vision and structure their income model accordingly. We think that this is a problem FinCare will encounter in the near future.

Dealing with communication and collaboration The risk of centralization in group-based collaboration networks and the risk of heterogeneous communication all have to do with communication and collaboration problems. The main mechanism underlying the first risk is the

amount of bottleneck in the development process. If there is one person sitting on the shared codebase, work on shared components needs to pass through that person first. If there is an issue with shared components, you need to go to that person. Depending on the responsibilities such a person has, this can hamper the development process of all projects.

For FinCare this mechanism is only partly relevant, since the work is currently comprehensible. However, it can become a high risk in the future because of the mechanism underlying the second risk: communication about updates. At FinCare updates on components just happen and there is little communication about them. Updates are done by this central person, hence there is little knowledge about what is really happening on the shared codebase. We therefore think that FinCare should involve more people in the development of the shared codebase. So a social self-regulating system of the codebase, with a central authority (or gatekeeper or champion) with the role of expert and quality manager.

§ 7.2 THINGS YOU ALREADY CAN IMPLEMENT TOMORROW

The answer to the problem statement is quite long, so what are the things both Topicus and FinCare can already do tomorrow? What are the ‘low hanging fruits’? First of all, we think that most issues are currently comprehensible for FinCare. But we do see these risks already of relevancy for the interviewed units. And since these units are larger than FinCare, we think that when FinCare grows larger and when their codebase increases in size, the issues will become more relevant. Implementing all recommendations at once might not be feasible, but the following three solutions are easy to implement and can already be done tomorrow:

Involve more people in the development of the shared codebase Currently the development of the shared codebase is centralized. Involving more people is easy to do and has a number of benefits. First of all, fresh insights and ideas are brought in when other people are involved in the development of the shared code. Secondly, there is more knowledge sharing of shared components if more people are involved in the development. Thirdly, if the central person drops out or is unavailable, the work is not compromised.

Have a simple, but clear communication policy when modifying components Currently, when components are updated, there is little communication. Having simple rules as to what to communicate, to whom and when increases the opportunity developers have to take the impact of the modifications into account.

Incorporate (elements of) feature-driven development Currently, teams at FinCare follow a project-driven development approach. Without documentation policies, active reuse opportunity seeking and communication of codebase enhancements there is little knowledge sharing between teams. Therefore we think that occasionally having developers and analysts switch to other project teams may already improve this.

§ 7.3 THINGS TO IMPLEMENT IN THE LONG TERM

Some recommendations require additional research before they can be implemented. We think the following recommendations should be implemented in the long term for both FinCare and Topicus.

Component development model Within business units employ a social self-regulating codebase, without strict code ownership, where everyone can contribute and is responsible for their own modifications. Do this by:

- Giving everyone read and write access on all components.
- Using strict versioning for all components. For each version change:
 - Send an update to all developers when a component or package is updated to a new version.
 - Plan updates on important components a week in advance and send an email to everyone to inform about the upcoming update.
- Involving more people in the development of shared components. A single person should not have all the responsibility over components.
- Strive for complete unit test coverage for important components.
- Monitoring of changes to the shared codebase and feedback by senior developers.
- Giving all developers and analysts the option to switch projects every few months.

Between business units, follow a producer-consumer paradigm. Units should give read access rights to other units of their components. This means that all units should use the same repository technology (either GIT, SVN or Mercurial). Producers of components should always design for reusable components and communicate changes to consumers. Consumers themselves are responsible for using some component and updating it.

Platform billing model Interviewees indicated that internally billing of work on a shared codebase can be unfair and hamper refactoring/investing in the shared codebase. Externally, to customers, this work effort is difficult to bill in a transparent way. The rationale behind this is as follows:

- From the perspective of the first customers, a starting-up SaaS solution is often a tailored solution
- Delivering software using a SaaS at some point requires a transition from tailored-offering to mass-offering
- With a small customer base, new features or refactoring effort on platform components are initiated from desires from a single or just a few customers. Customers bargaining power then is high. Platform effort is directly billed to customers, since other customers do not want to pay for features they don't need.
- With a large customer base, supplier bargaining power is much higher. Hence, the supplier can follow its own product road map. New features are added based on customer feedback and made available through a standardized pricing model.
- Depending on the stage of the SaaS solution, different licensing models are offered to customers.
- In all cases developed features or improvements must be paid for. Depending on the development stage of the SaaS solution, work effort is then either paid directly from the licensing fees, from the contract with individual customers or from an upfront investment by the supplier.
- If there is no agreement among product teams in the earlier stages of platform development as what kind of efforts is paid by the company (making components reusable for instance) and what is covered by licenses, refactoring and making components reusable is hampered. Also, if different products follow different licensing models, internal hour billing can be unfair if for one team these costs are covered by the license, whereas for other teams this is not covered at all. Why would such a team contribute to the shared codebase?

Therefore we think that this requires a Topicus wide consistent component development model which should include the following elements a number of standardized licensing models for SaaS products for the different stages a SaaS product can be in. Important here is that these licensing models should show the difference between custom work and general platform effort. Custom work should never be included in the standard license fee, since other customer don't want to pay for features they don't have or get. First of all, standardized licensing models should make it easier for business units to acquire customers, since you can make a more stronger case with transparent and proven business models. It prevents units to reinvent the wheel. Secondly, it makes billing of work effort to the shared codebase more transparent and fair, both to customers as among different teams.

Component usage visualization A very important knowledge question which came up during the interviews and during the case study at FinCare is what applications use what component. For our case study at FinCare we developed some small tools to visualize the interdependency between applications and components. We strongly think that these kind of visualizations can help developers and analyst quickly see who is using what component. This is useful when either analyzing the impact of some modification, or when looking for information regarding a component. Future research on what level of view is of most value for developers and analyst, finding out what the best way is to technically implement this can be very valuable for Topicus.

Feature-driven development We state that adopting elements of feature-driven can increase knowledge sharing and overall quality of a shared codebase. However, feature-driven development can be implemented in numerous shapes and forms. Future research is needed to find out what the best practices are, what the best way is to implement this paradigm within Topicus.

§ 7.4 SCIENTIFIC RELEVANCY

With respect to the scientific relevancy of this study, first of all, in chapter 3 we conducted a literature study of case studies with respect of working in a shared codebase environment. The list of 28 risks we identified is in our opinion a very valuable addition to the challenges as identified by (Ghanam et al., 2012). On this topic there is little scientific material available describing actual practical experiences. Our list, based on a wide variety of good industrial studies, can be used for companies to understand the issues and mechanisms underlying working with and on a shared codebase.

Secondly, also of scientific relevancy are the results from the interviews. We conducted 8 semi-structured interviews at 3 different business units of Topicus. From this we abstracted in-depth issues and best practices on a business, organizational, process, people and technical level. For software companies with a comparable agile development culture, the best practices and described experiences from the interviewees give insight in how people work and deal with every day issues.

Thirdly, we showed that in a shared codebase environment bottom-up code visualization can be very valuable. This kind of analysis can give insight in the complex interdependency of software assets on a level of abstraction relevant to find out what applications are using some component. This is a knowledge question developers and analysts at Topicus indicated is very difficult to answer with the current development tools available.

§ 7.5 VALIDITY

Having stated the conclusion we will discuss some points which may affect the validity of this study. The first point is the completeness of the risks we have identified from the literature study. The risks have been identified from industrial case studies published in the period of 2000 and 2012. Hence, we excluded any ground theoretical studies published on the topic of product-line engineering, which was a very active research field in the period 1990-2000. At the time we did the literature study, we did not know that, but nevertheless we think the list of risks was sufficient for the purpose of this study. Also, only having included recent material makes the results more relevant for a young company like Topicus. The only thing we know we really missed are issues with respect to having a SaaS delivery model.

The second validity point is the unequal spread of interviewees from the business units. From unit A we interviewed 2 people, 5 from unit B and 1 from unit C. This may have caused the ideas from unit B to be more prominently all over the page, because there simply is more material from that unit. Still, diversity among the interviewees was high (different roles, different projects) and we did follow a semi-structured approach so we adjusted questions depending on what we heard during the interview. Also, we think that each interview provided new insights and since we don't draw any conclusions based on the number of people mentioning some aspect, the results are valid enough for this study.

The third validity point has to do with the creation of the shortlist of risks. We did not conduct any formal interviews within FinCare, rather we based the risk analysis on the experiences or observations we gained by just being around. We sat down a couple of times for some explanation about the setup of the codebase, but we don't have any empirical data to support the determination of impact, likelihood and relevancy besides our own observations and the results from the bottom-up codebase analysis.

The fourth validity point also is about the shortlist. We created a shortlist of risks based on impact and likelihood which combined determined the relevancy of some risk for FinCare. In the discussion on each risk in section 5.3 we try to explain why a risk has a 'high' or 'very low' impact or likelihood of occurrence and how this relates to relevancy, but it remains a bit arbitrary, since it mainly is a subjective approach.

The fifth validity point is that the problems and solutions from the interviews are not directly related to requirement changes. We think a large number of these problems and solutions are more general challenges of working on a shared codebase. We tried, by relating them to the risks from the literature study, to filter for those solutions which are related to volatility of components caused by changes. Nevertheless, the line between what is a general challenge with respect to working on shared codebase and what is a risk of changing requirement is very thin.

The sixth validity point is that we state that most risks currently are comprehensible for FinCare, but that the relevancy may increase when FinCare and their codebase grows larger. In our introduction we briefly stated that Topicus follows a strategy where units are encouraged to split up when they grow larger than 25 people. When FinCare is split up in the near future it is arguable that a lot of these risks will become of lower relevancy because the scope of the work, again, becomes surveyable. We argue that while this may be the case, this will only work for the FinCareAnalyse product, since they are relatively uncoupled from the other applications. All products depending on the FinCareClaim platform on the other hand are strongly coupled, hence the risks will stay highly relevant.

Finally we think that, despite these validity points, our recommendations still make a strong case. They come directly, first-hand, from experiences from within the organization. They represent problems which are not uncommon in the field of software engineering, about which little scientific data is available so where learning about the latest best practices is very valuable.

Appendices

- Appendix A -

Challenges versus change characteristics

	business strategy	instability	dominance of a mainstream product	competing goals	among platform teams	between platform teams and application teams	in distributed development	between business units	silos	decision-making	stakeholder involvement	feature versus component teams	team autonomy	business-value thinking	product ownership thinking	agility versus stability	of documents	of practices	of tools and technical solutions	of acceptance criteria	reuse	variation sources	cross-cutting concerns	different actors	requirement of combination	requirement of maximizing reuse	Accessibility	Platform quality	Platform stability	Testing	Continuous integration	Release synchronization
time of change/discovery activity	1		1										1		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	18	
change type/category		1																			1	1	1	1	1	1	1	1	1	1	7	
change history											1																					13
Frequency		1			1	1	1	1	1	1	1				1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	16	
trigger/source	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20	
domain	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20	
phase	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	25	
manager's control	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	23	
# stakeholders																															0	
cost	1	1												1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	8	
value	1	1												1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	8	
Motivation/opportunity					1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19	
description																															0	
criticality	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	26	
developer experience	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20	
granular effect	1	1	1	1	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22	
properties																															0	
features																															0	
quality attributes	8	9	6	6	7	7	7	7	7	10	5	1	5	10	9	7	9	4	4	4	13	7	10	5	9	8	8	9	9	11	12	0

Figure A.1: Reuse challenges vs change characteristics

- Appendix B -

Interview protocol

Describe current position

1. How would you describe your current position?
2. In what projects are you involved?
3. What is your role in these projects?

Describe perception of a requirements change

4. How would you describe a requirements change?

Shared functionality

5. For the projects you work on, is there a shared codebase and how is it used?
6. Can you explain the inter-dependencies of the components and products of the codebase?

The process of handling a changing requirement

7. Could you describe how the development process looks like?

Dealing with changes on shared components

8. What problems do you encounter when implementing changes on the shared codebase?

Testing

9. Could you describe how products are tested?
10. If you would have to estimate, how much unit-test coverage is there?

Knowledge sharing

11. Do you think the amount of knowledge sharing is sufficient?

Documentation

12. What kind of documentation is used during development?

Chosen solutions

13. How do you mitigate problems encountered during development on a shared codebase?

Recommendations

14. What would you like to see differently with respect to working on a shared codebase?

Reuse challenges

- | | |
|----------------------------|----------------------------------|
| 1. Reuse | 8. Instability |
| 2. Continuous integration | 9. Product ownership thinking |
| 3. Release synchronization | 10. Standardization of documents |
| 4. Testing | 11. Requirement of combination |
| 5. Cross-cutting concerns | 12. Platform quality |
| 6. Decision-making | 13. Platform stability |
| 7. Business-value thinking | 14. Business strategy |

Change characteristics:

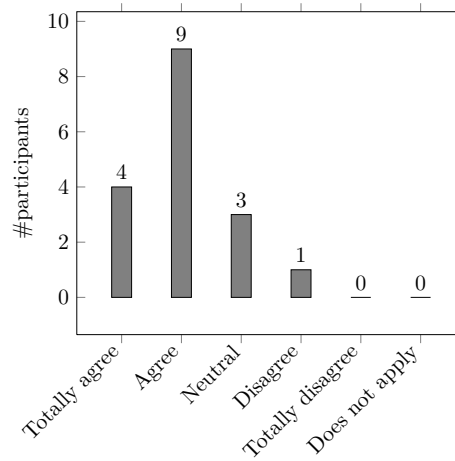
- | | |
|----------------------|---------------------------|
| 1. Criticality | 6. Domain |
| 2. Project phase | 7. Developer experience |
| 3. Manager's control | 8. Motivation/opportunity |
| 4. Granular effect | 9. Time of change |
| 5. Trigger/source | 10. Frequency |

- Appendix C -

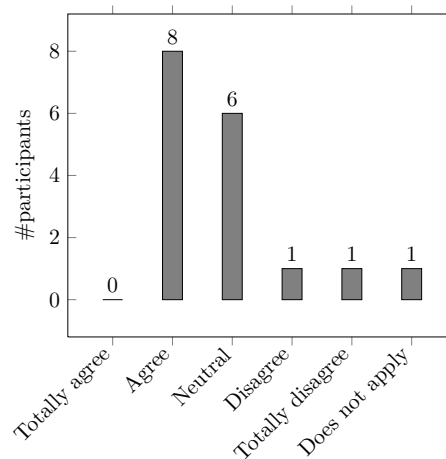
Questionnaire

We distributed a questionnaire at FinCare containing 23 statements about the problems and solutions as presented in section 6. We asked everyone at FinCare to rate the statements according to a 5 point Likert scale. Below the results are displayed.

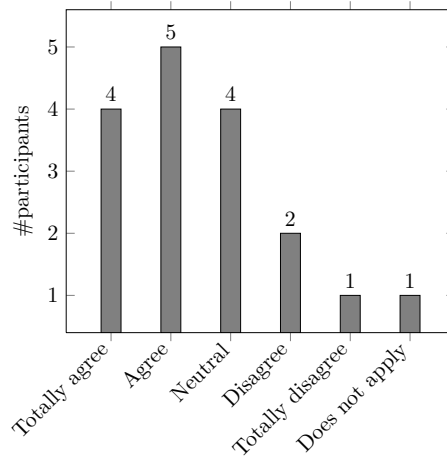
Statement: Versioning components is a good approach to assure component stability.



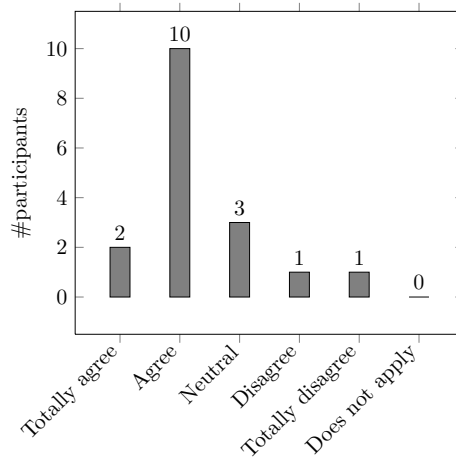
Statement: Striving for complete code coverage of shared web services is a good approach to prevent unintended effects in web services in other applications.



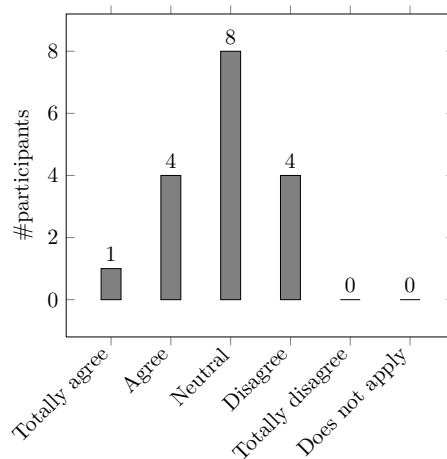
Statement: Striving for complete code coverage of unit tests for core components is a good approach to assure stability.



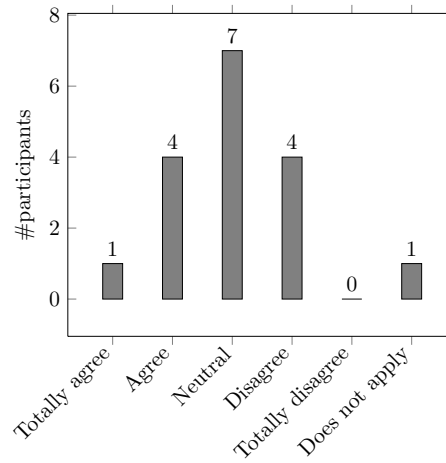
Statement: When modifying a component I'd like to know better what applications use this component.



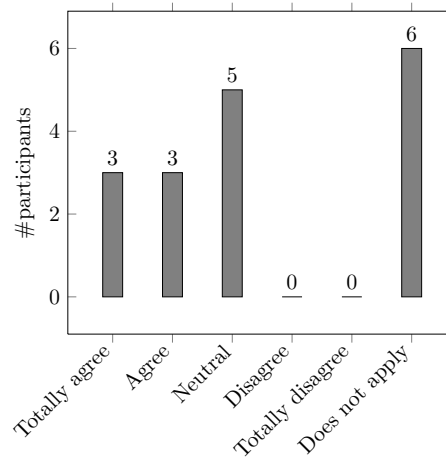
Statement: I would like to see a more positive attitude with respect to reusable components.



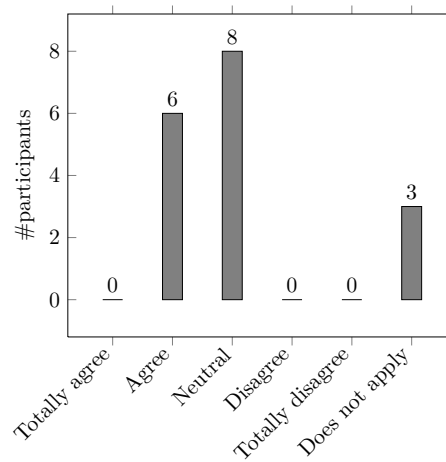
Statement: Enough time is available for personal initiatives to improve components.



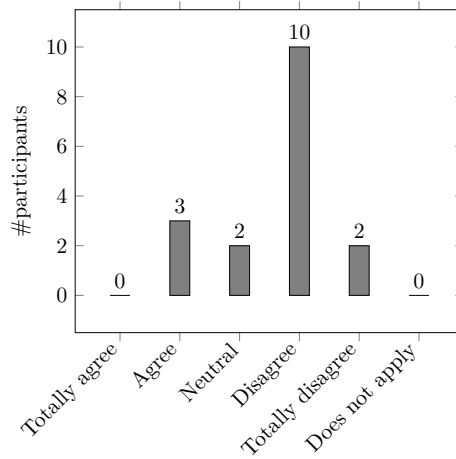
Statement: I can honestly justify hours spent on components.



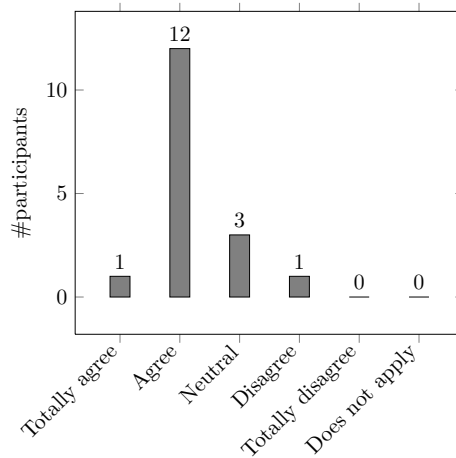
Statement: Towards customers hours spent on components can be justified in good fashion.



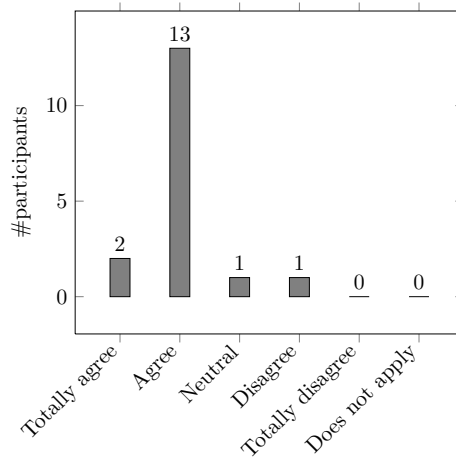
Statement: Everyone should be able to modify components as their own responsibility.



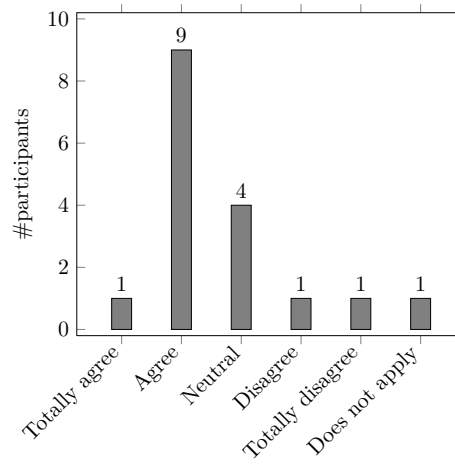
Statement: A contact person for each component is required in order to have clear lines of communication.



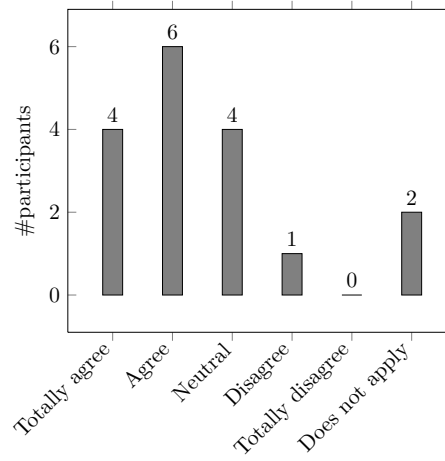
Statement: A simple but clear communication policy for component updates would be an improvement.



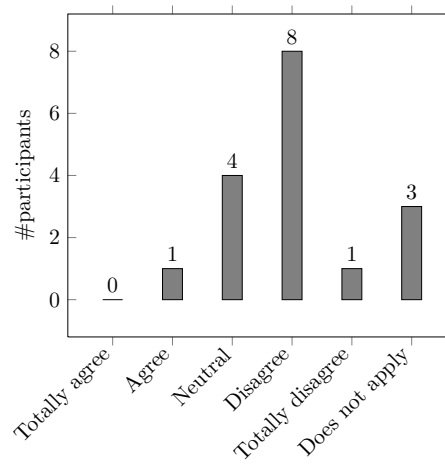
Statement: There is enough awareness about the responsibilities of working with components.



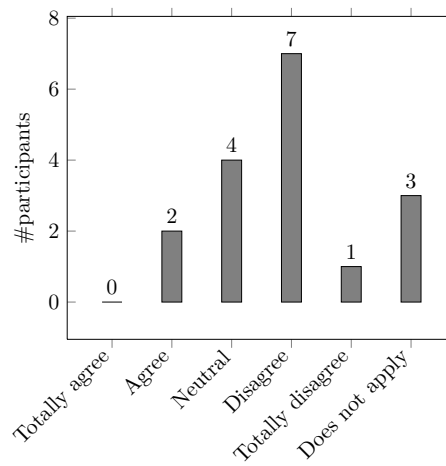
Statement: Components which I use are often updated by others outside my project team.



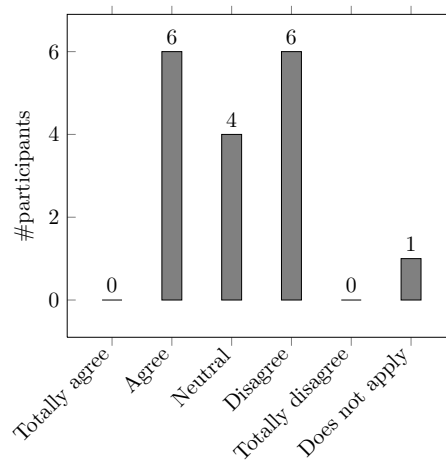
Statement: When I push a modification to a component I'm certain that no defect is introduced in another application.



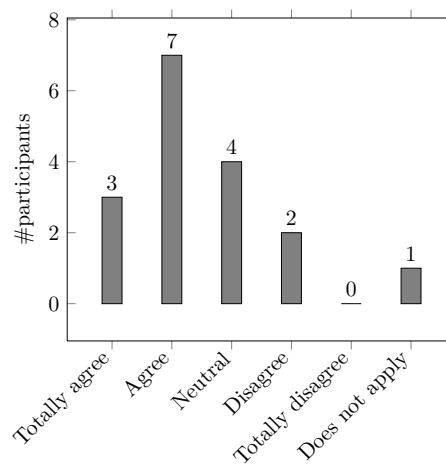
Statement: When I push a modification to a component I'm certain of the impact for other applications.



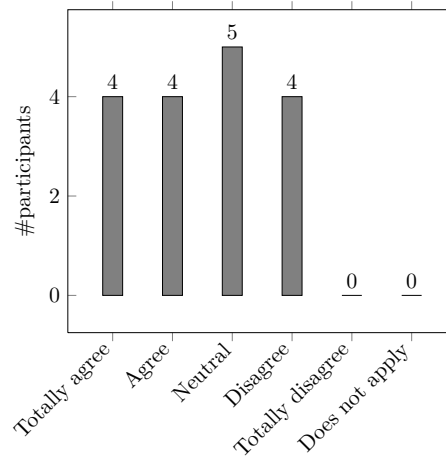
Statement: At FinCare a platform is put into production too soon, with too many open issues.



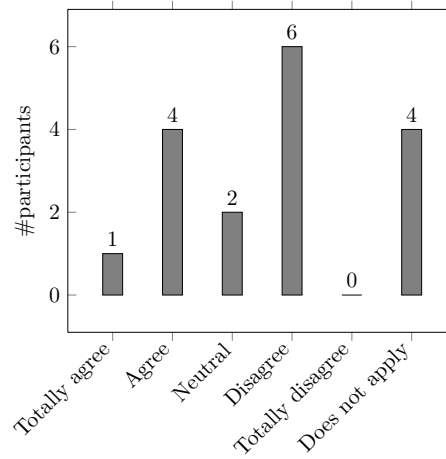
Statement: Components at FinCare are reusable and directly employable in other projects.



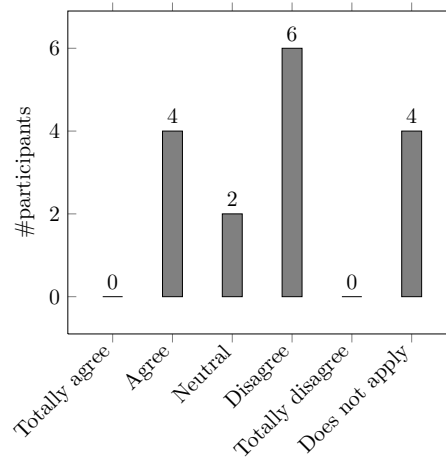
Statement: At FinCare refactoring and code cleaning is pushed forward or postponed a lot.



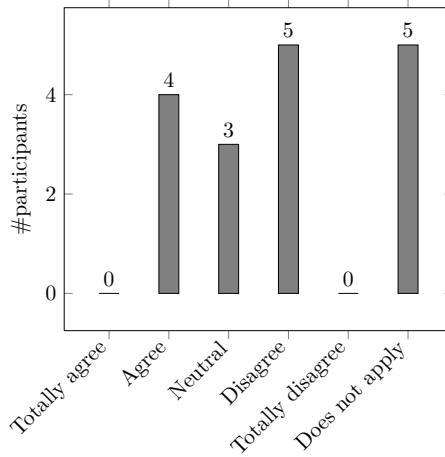
Statement: Modifications on components are always done by the same person.



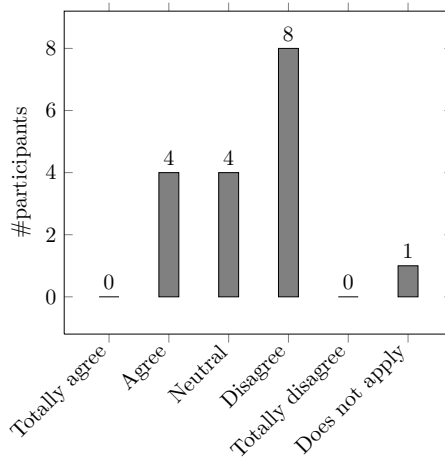
Statement: I know who to notify when I modify a component.



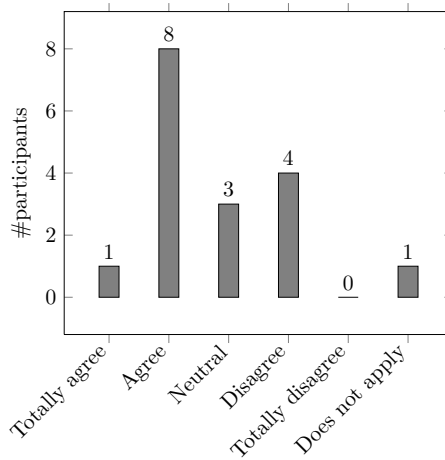
Statement: I always notify the correct people when modifying a component.



Statement: When a component is modified, people inform each other well.



Statement: When a component is modified I have enough time in advance to deal with the potential impact of the change.



Bibliography

- Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.
- Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An open source software for exploring and manipulating networks.
- Beckhaus, A., Karg, L., and Neumann, D. (2010). The impact of collaboration network structure on issue tracking's process efficiency at a large business software vendor. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE.
- Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Professional.
- Bosch, J. (2001). Software product lines: organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100. IEEE Computer Society.
- Breivold, H., Larsson, S., and Land, R. (2008). Migrating industrial systems towards software product lines: Experiences and observations through case studies. In *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, pages 232–239. IEEE.
- Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al. (2010). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM.
- Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332.
- Chapman, M. and van der Merwe, A. (2008). Contemplating systematic software reuse in a project-centric company. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 16–26. ACM.
- Charette, R. (1989). Software engineering risk analysis and management.
- de Jonge, M. (2005). Build-level components. *Software Engineering, IEEE Transactions on*, 31(7):588–600.

- de Souza, C. R., Quirk, S., Trainer, E., and Redmiles, D. F. (2007). Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proceedings of the 2007 international ACM conference on Supporting group work*, pages 147–156. ACM.
- Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., and Aho, A. (2008). Do crosscutting concerns cause defects? *Software Engineering, IEEE Transactions on*, 34(4):497–515.
- Faraj, S. and Sproull, L. (2000). Coordinating expertise in software development teams. *Management science*, 46(12):1554–1568.
- Ferreira, S., Collofello, J., Shunk, D., and Mackulak, G. (2009). Understanding the effects of requirements volatility in software engineering by using analytical modeling and software process simulation. *Journal of Systems and Software*, 82(10):1568–1577.
- Ferreira, S., Shunk, D., Collofello, J., Mackulak, G., and Dueck, A. (2011). Reducing the risk of requirements volatility: findings from an empirical survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(5):375–393.
- Fichman, R. G. and Kemerer, C. F. (2001). Incentive compatibility and systematic software reuse. *Journal of Systems and Software*, 57(1):45–60.
- Flick, U. (2009). *An introduction to qualitative research*. Sage Publications Ltd.
- Gall, H., Hajek, K., and Jazayeri, M. (1998). Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 190–198. IEEE.
- Garcia, V., Lucrédio, D., Alvaro, A., De Almeida, E., de Mattos Fortes, R., de Lemos Meira, S., and Recife, P. (2007). Towards a maturity model for a reuse incremental adoption. In *the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2007), Campinas, São Paulo, Brazil*. Citeseer.
- Ghanam, Y., Maurer, F., and Abrahamsson, P. (2012). Making the leap to a software platform strategy: Issues and challenges. *Information and Software Technology*.
- Gupta, A., Cruzes, D., Shull, F., Conradi, R., Rønneberg, H., and Landre, E. (2010). An examination of change profiles in reusable and non-reusable software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(5):359–380.
- Han, J. and Kamber, M. (2006). *Data mining: concepts and techniques*. Morgan Kaufmann.
- Han, W. and Huang, S. (2007). An empirical analysis of risk components and performance on software projects. *Journal of Systems and Software*, 80(1):42–50.
- Hanssen, G. (2011). Agile software product line engineering: enabling factors. *Software: Practice and Experience*, 41(8):883–897.
- Hanssen, G. and Fægri, T. (2008). Process fusion: An industrial case study on agile software product line engineering. *Journal of Systems and Software*, 81(6):843–854.
- Kagdi, H., Maletic, J., and Sharif, B. (2007). Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 145–154. IEEE.

- Larman, C. and Vodde, B. (2009). Scaling lean & agile development. *Organization*, 230:11.
- Lehnert, S. (2011). A review of software change impact analysis. <http://www.db-thueringen.de/servlets/DocumentServlet?id=19544>. Accessed August 20, 2012.
- Lindlof, T. and Taylor, B. (2002). *Qualitative communication research methods*. Sage Publications.
- Lucrédio, D., dos Santos Brito, K., Alvaro, A., Garcia, V., de Almeida, E., de Mattos Fortes, R., and Meira, S. (2008). Software reuse: The brazilian industry scenario. *Journal of Systems and Software*, 81(6):996–1013.
- McGee, S., G. D. (2011). Software requirements change taxonomy: Evaluation by case study. pages 25–34.
- McGee, S. and Greer, D. (2009). A software requirements change source taxonomy. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 51–58. IEEE.
- Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society.
- Mohagheghi, P. and Conradi, R. (2008). An empirical investigation of software reuse benefits in a large telecom product. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3):13.
- Muffatto, M. and Roveda, M. (2002). Product architecture and platforms: a conceptual framework. *International Journal of Technology Management*, 24(1):1–16.
- Munkvold, B., Eim, K., and ØYVIND, S. (2006). A case study of information systems decision-making: Process characteristics and collaboration technology support. *International Journal of Cooperative Information Systems*, 15(02):179–203.
- Noor, M., Rabiser, R., and Grünbacher, P. (2008). Agile product line planning: A collaborative approach and a case study. *Journal of Systems and Software*, 81(6):868–882.
- Nurmuliani, N., Zowghi, D., and Powell, S. (2004). Analysis of requirements volatility during software development life cycle. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 28–37. IEEE.
- Otsuka, J., Kawarabata, K., Iwasaki, T., Uchiba, M., Nakanishi, T., and Hisazumi, K. (2011). Small inexpensive core asset construction for large gainful product line development: developing a communication system firmware product line. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 20. ACM.
- Perry, D., Siy, H., and Votta, L. (2001). Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337.
- Pohl, K., Bockle, G., and Van Der Linden, F. (2005). *Software product line engineering*, volume 10. Springer.

- Ponte, D., Rossi, A., and Zamarian, M. (2008). The role of competencies and interests in developing complex information technology artefacts: The case of a metering system. *Open IT-Based Innovation: Moving Towards Cooperative IT Transfer and Knowledge Diffusion*, pages 291–308.
- Ramasubbu, N. and Balan, R. (2010). Evolution of a bluetooth test application product line: a case study. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 107–116. ACM.
- Rothenberger, M. (2003). Project-level reuse factors: Drivers for variation within software development environments*. *Decision sciences*, 34(1):83–106.
- Schröter, A., Aranda, J., Damian, D., and Kwan, I. (2012). To talk or not to talk: factors that influence communication around changesets. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1317–1326. ACM.
- Scott-Morton, M. (1991). The corporation of the 1990s: Information technology and organizational transformation. *Sloan School of Management, Oxford University Press, New York*.
- Sherif, K. and Vinze, A. (2003). Barriers to adoption of software reuse: A qualitative study. *Information & Management*, 41(2):159–175.
- Slyngstad, O., Li, J., Conradi, R., Ronneberg, H., Landre, E., and Wesenberg, H. (2008). The impact of test driven development on the evolution of a reusable framework of components—an industrial case study. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 214–223. IEEE.
- Stuckenholz, A. (2005). Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes*, 30(1):7.
- Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software: beyond object-oriented programming*. Addison-Wesley.
- Tang, A., Couwenberg, W., Scheppink, E., de Burgh, N., Deelstra, S., and van Vliet, H. (2010). Spl migration tensions: an industry experience. In *Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering*, page 3. ACM.
- Van der Linden, F. (2001). *Software Architectures for Product Families: International Workshop IW-SAPF-3. Las Palmas de Gran Canaria, Spain, March 15-17, 2000 Proceedings*, volume 1951. Springer.
- Van Gorp, J. and Bosch, J. (2002). Design erosion: problems and causes. *Journal of systems and software*, 61(2):105–119.
- van Gorp, J., Prehofer, C., and Bosch, J. (2010). Comparing practices for reuse in integration-oriented software product lines and large open source software projects. *Software: Practice and Experience*, 40(4):285–312.
- Wallace, L., Keil, M., and Rai, A. (2004). Understanding software project risk: a cluster analysis. *Information & Management*, 42(1):115–125.
- Webster, J. and Watson, R. (2002). Analyzing the past to prepare for the future: Writing a literature review.

- Williams, B. and Carver, J. (2010). Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51.
- Wolfswinkel, J., Furtmueller, E., and Wilderom, C. (2011). Using grounded theory as a method for rigorously reviewing literature. *European Journal of Information Systems*.