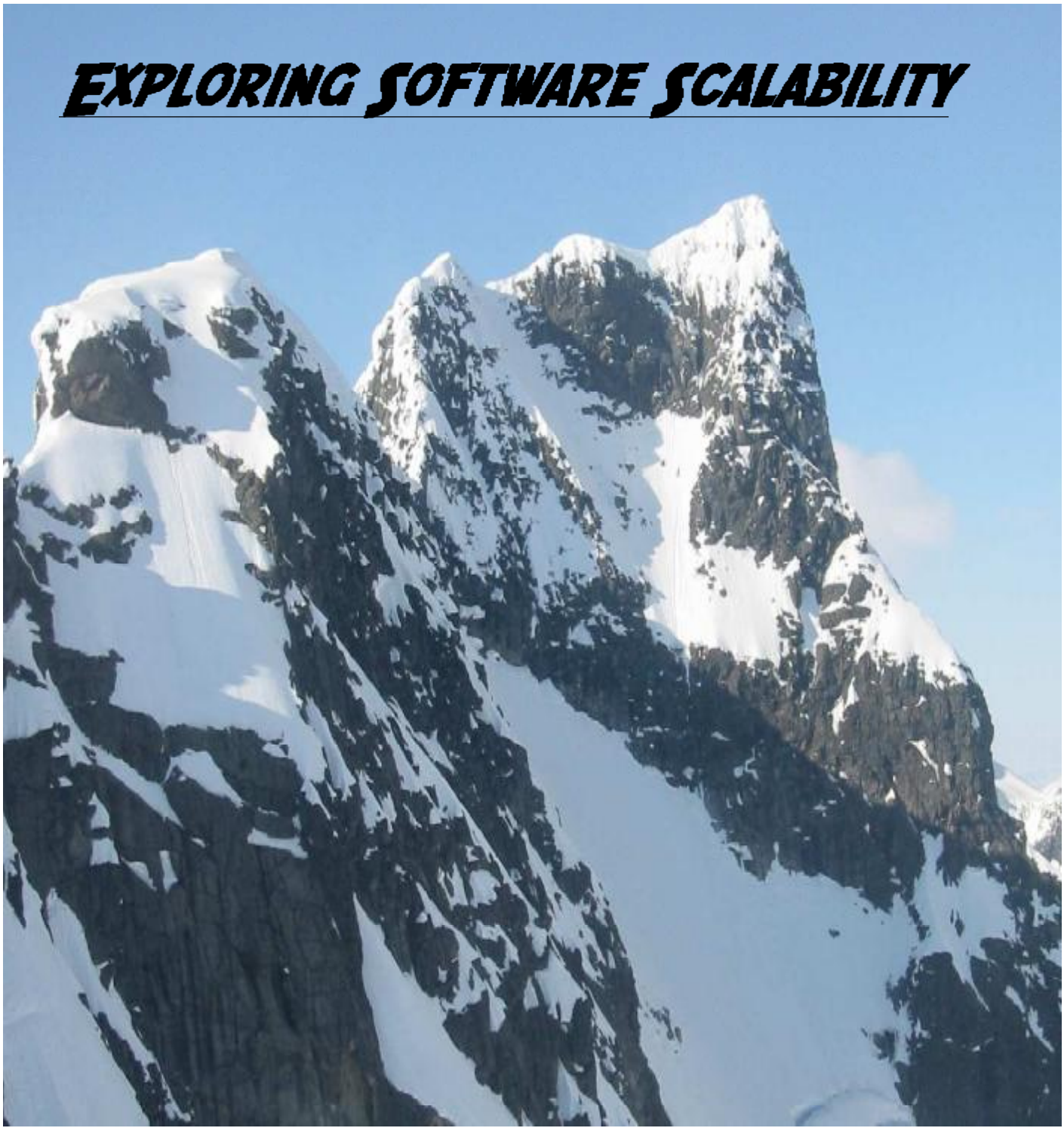
Exploring Software Scalability



topicus

University of Twente
Enschede - The Netherlands

Cover

The mountain on the front is the Store Austanbotntind a well known mountain in Norway. With a little above 2.200 meters it is one of the highest mountain peaks of Norway. I choose this cover because it represent two important subjects of my thesis, scaling and exploring. Trying to climb a mountain is as much about mental as physical preparation. A good climber will combine both and thus successfully explore and scale a mountain. This thesis also wants to combine all necessary steps and preparations to be able create and maintain scalable software.

Exploring software scalability

And a method for evaluating and improving software scalability

Thesis submitted for the degree
of Master of Science at
the University of Twente

Thijs Pieter Munsterman

Enschede, February 2008

Graduation Commission
Dr. M. Daneva (UT – EWI faculty)
C. Amrit MEng (UT – IVB faculty)
Ir. M. Krans (Topicus)
Ing. R de Negro (Topicus)

topicus

University of Twente
Enschede - The Netherlands

Management Summary

Scalability is an important research topic in the current age of Internet enabled applications. From an academic viewpoint there is not much research about scalability. As a consequence it is a practical problem for organizations to measure or predict the scalability of (parts of) their systems. This research focuses on scalability on the software side of a system.

Current scalability literature is characterized by two things. One, there is no widely accepted definition of scalability. Almost every researcher uses his/her own definition and only looks at scalability from a single perspective. Two, this perspective is mostly about the relationship between hardware and scalability. This lack of existing knowledge about software scalability generates two goals for this thesis. The first being a definition of software scalability. What is software scalability and when can it be used? The second being a case study about analyzing and improving software scalability.

Software scalability is defined as *the ease with which the software of a system can be expanded to serve more users and/or work*. This definition is the overall goal of software scalability. The main advantage of software scalability, in contrast to hardware, is that the solutions is reusable during different situations. This definition can be further divided into two dimensions:

1. *Software scalability is the ability to handle increased workload by changing parts of the code (scalability optimization/vertical scalability)*
2. *Software scalability has the ability to be used multiple times in a cost-effective way (environmental flexibility/horizontal scalability)*

The two dimensions are an important part of software scalability and should be both be considered when analyzing software scalability. Both are equally important because both are important for creating scalable software. Both might require different things of an application a trade-off decision has to be made about which aspect is more important in a specific situation.

To assist an organization in achieving software scalability an evaluation method is proposed. The method is build around three key steps, *finding bottlenecks, analyzing a bottleneck* and *solving a bottleneck*. During the different steps a number of well-known and easy usable tools, methods and metrics are used to either find or analyze software scalability. A case study was done at Topicus to evaluate the method. As a developer of web applications Topicus has a vested interest in creating scalable software. The case study was successful in finding and analyzing a bottleneck in a Topicus application.

Not all the selected tools, method and steps were usable to analyse the scalability dimensions. Especially horizontal scaling proved to be difficult to measure. The ability to reuse code in many different situations is dependent on a lot of different aspects. Five different metrics were chosen to quantify the horizontal scaling ability. However the results of the metrics remained inconclusive about wether or not they were truly indicative of scalability. Future horizontal scaling needs a better metric that better fits the goal of horizontal scaling.

In the end software scalability is only one side of scalability. True scalability is a synergy between hardware and software. Both have their own place and offer different things and should be considered as such. Improving a system along all sides and dimensions makes a system future proof from several angles.

Acknowledgements

This thesis is the final result of a seven month research project. The research project is the final step of my university graduation but also entire education part of my life. After my graduation I will take my first steps into the IT sector as a full time employee.

I want to thank a lot of people who helped my during my graduation phase in different ways. First I would like to thank my supervisors from both the university, Maya Daneva and Chintan Amrit, and Topicus, Martin Kransen Rob de Negro for their continual input and feedback on my research and for answering all my questions. I also want to thank Topicus in general and the Finance department in particular for creating the opportunity for this research. During my time in your office I learned a lot about their unique approach to building good software but also creating a good work environment. I had a lot of fun during all the social activities.

I also want to thank Hidde, Johan, Jonathan, Mirjam and Ted in person for all the additional help they gave during these months.

Thijs Pieter Munsterman
Thursday, 07 February 2008
Deventer

Table of content

List of figures

List of tables

1. Introduction

This chapter discusses the central problem of this thesis. The following sections explain the problem, its background and the way this problem is currently tackled. The chapter continues with setting out the research goal of this master project, outlining the research method, and formulating the research questions.

1.1 Problem background

The quality of software is a hot topic in the software industry. The growth of the IT industry brought forth many different initiatives proposed to assess and improve the quality of software [29]. In today's competitive market, the ability to deliver high quality software, while keeping the resource demand within budget has become an important competitive advantage [14]. Especially in e-Business the quality of the application is believed to be of high importance to its competitive advantage [13], [21], [45].

A paper from 2001 asked an organization to identify and categorize potential risks for a traditional IT project and for an e-Business project [21]. For traditional IT projects most risks were linked to functionality. For the e-Business project the functionality of the application was no longer the most important risk. Different quality related risks were identified to be just as, sometimes more, important as functionality. The results are shown in Figure 1.1.

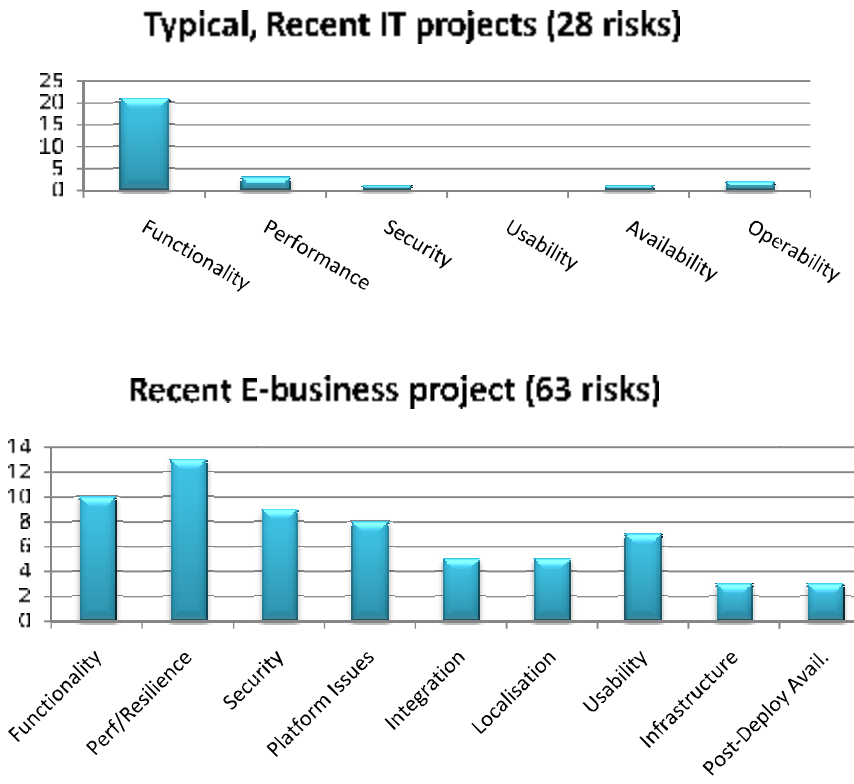


Figure 1.1: Result from survey of risk of an IT and e-Business project [from [21]]

The research project started with a short preliminary study about the most critical non-functional requirements for web application (see Appendix A: Preliminary investigation into Non-functional requirement). When developing software for the web, one of the environment variables that has to be taken into account is the number of expected users and the load they put on the application. The ability of an application to handle the increased load is defined as scalability. A perfectly scalable application should be able to handle an increased load with minimal increase in resource demand and a minimal decrease of performance. While scalability is used to describe a certain quality aspect of the software it is often poorly understood and defined [26]. The reason scalability is poorly understood is its inherent complexity. Scalability is a multi-faceted problem that has been poorly researched.

The danger of this gap in knowledge is that it makes it hard to predict the scalability. The scalability of an application is hard to precisely predict before it is actually created. An organization can improve its knowledge about scalability by reusing (parts of the) code that has been implemented and tested for scalability in the past. There is however no real research about the implementation of software scalability and its reuse.

Reusing implemented code can cause difficulties because the previous environment may not match the new environment. Differences in the environment can cause undesirable effects in the software code, which can lead to faults and errors. Components are made more generic to enable implementation in multiple environments. If a component has been designed without scalability in mind it is hard to predict the scalability of the result. This problem is shown in Figure 1.2.

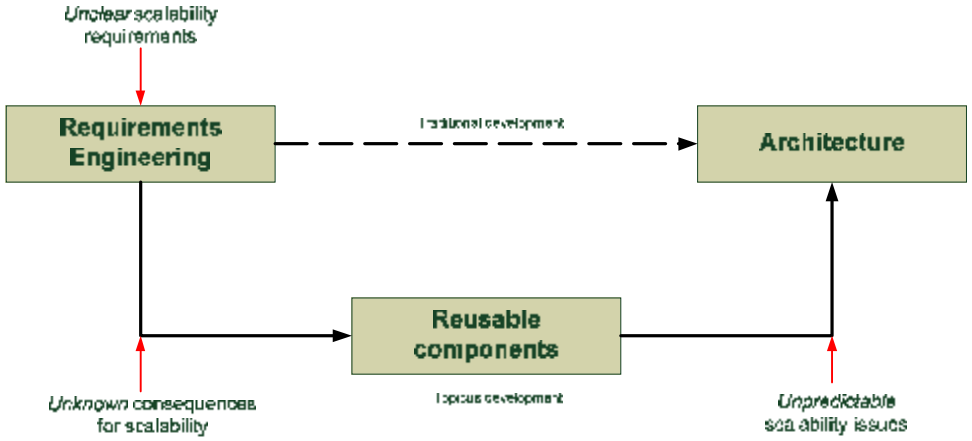


Figure 1.2: Problem background

The main issue of software scalability is the lack of coherent and directly applicable knowledge about the subject. Gaining knowledge about software scalability and developing implementable practices can help an organization in successfully improving scalability of its application and improving the competitive advantage of the product and the organization.

1.2 Problem statement

The complexity of the scalability issue is an important starting point for the issue of evaluating scalability. The desire to have a better insight into scalability leads to the following problem statement:

"How can the software scalability of an application be analyzed for the goal of improving the scalability of the application and gaining knowledge about the scalability issue?"

1.3 Research goal

The goal of this research is to investigate options for improving the current scalability analysis practice as part of web application development projects. The first goal is to define and explore the concept of software scalability and how it is related to the broader concept of scalability, and other quality requirements. This goal is motivated by the fact that in literature software scalability is overshadowed by the research and application of hardware scalability.

The second goal is to create a roadmap that provides the ability to improve the software scalability of applications and gather knowledge about scalability. This is motivated by the experiences of team members at Topicus, who observed that the ability to assess the scalability of an application increases the chance that the resulting scalability is acceptable.

To achieve these research goals an exploratory study on the subject of software scalability has been done. The used research method is described in more detail in subsection 1.7.

1.4 Research model

The research consists of a number of steps that combine both theoretical and practical approaches. The different research goals are depicted in one or more steps, that combined provide an answer to the larger goal. The theoretical part is the creation of an evaluation model for software scalability. The newly created model is applied and analyzed in a case study using software provided by Topicus. Information gathered this way is used for the evaluation of the model.

The final step is the implementation and evaluation of the model in an organizational context. The exploration of software scalability can only be truly successful if it is supported over a larger period of time. The research model is depicted in Figure 1.3. It was developed by applying Verschuren's and Doorewaard's method for composing a research model [55]. This research model combines all these steps and puts them in a logical order.

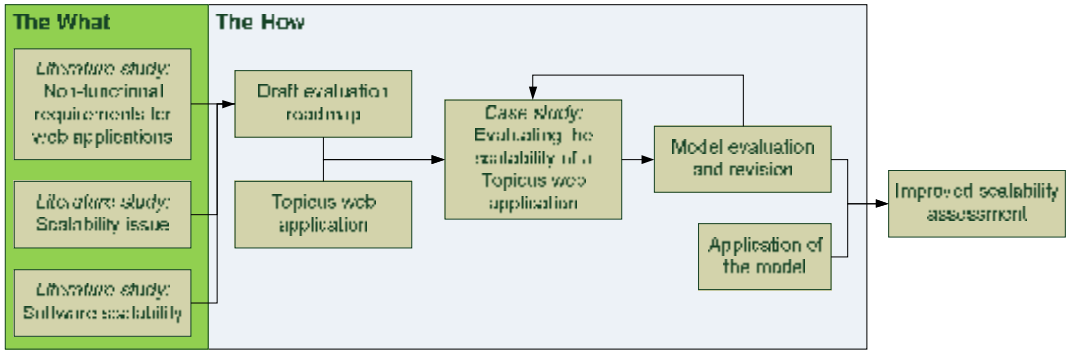


Figure 1.3: Research model

1.5 Research questions

The research model in Figure 3 is operationalized by research questions. The separate steps of the research model can be viewed as questions to be answered. Four questions were identified that were answered during the project, in order to satisfy the main goal. The following questions are of importance for providing an answer to the central question of the project:

1. *What is scalability, how is it defined in literature and what are typical scalability issues?*
   Different authors talk about the meaning of scalability, the importance of scalability and typical scalability issues. A literature survey is needed to provide insight the current state-of-the-art of scalability.

2. *How can software scalability be defined?*
   Scalability in software has no real unifying definition. A new usable definition is needed to capture the meaning of scalability for software.

3. *What can be done for the continued evaluation and improvement of software scalability?*
   Three basic steps are proposed to ensure a continues improvement of scalability:
   (1) bottleneck identification, (2) bottleneck analysis and (3) bottleneck solution strategy.
   For scalability this means the first step is finding and identifying the most important scalability issue. The second step is analyzing the problem in order to be able to link cause and effect. The third step is choosing an effective solution strategy to tackle the problem.

4. *Is the evaluation method usable?*
   The evaluation and improvement of scalability is not a onetime event or practice. Continues assessment and information gathering can provide valuable knowledge for the organization. Which knowledge is usable and what should an organization do with this knowledge

1.6 Research scope

The focus of this research is on software scalability
While scalability is influenced by a great number of factors from both hardware and software, this research focuses on the scalability of software. Within software the focus is on the application code. Other software parts, such as databases and the operating systems, are not evaluated.

The impact of the environment is assumed to be minimal
The focus is on software, the influence of hardware or other software parts on the scalability of an application is assumed to be minimal. This research only proposes to make changes to the application, and not other factors as the hardware or the operating system.

Only a small numbers of evaluation methods is investigated
The purpose of this research is to investigate the desirability and outline for the evaluation of software scalability. For different situations, different methods and tools may be more applicable. This research is not interested in creating an overview or a survey of applicable methods and tools. Instead it merely focuses on the goal of the evaluation, not the precise method.

Interaction between non-functional requirements is assumed to be minimal
In a real-life environment there is interaction between different non-functional requirements. This interdependency requires a trade-off decision. For the sake of simplicity the interaction between scalability and other non-functional requirements is assumed to be minimal.

1.7 Research method

This research project is divided into three stages. The first stage is gathering information about the concept of (software) scalability and related areas. A literature study is done to define scalability in general and software scalability in particular. Which factors are of importance for achieving software scalability, and what are the potential benefits for improving software scalability?

The second stage is creating and using the new roadmap for evaluating, analysing and improving software scalability. A case study is done with an actual web application. For this research the web application is provided by Topicus. The goal of the scalability analysis is twofold. The first goal is evaluating the current scalability and identifying the scalability bottlenecks of the application. The second goal is generating knowledge about scalability. Which parts work well and which show (potential) problems?

The last stage is the evaluation and analysis of the roadmap and its applicability in the larger context. Is the roadmap usable for continual evaluation of scalability? Does the roadmap improve scalability and is it usable in practice? One of the subjects of the evaluation will therefore be the usability of the model in different environments. To truly create an environment where there is focus on quality proposing a single method is not enough. Quality has to become an important issue during the entire development cycle. To further increase knowledge about scalability in the organization, the awareness of scalability has to become part of the organization. Scalability is a moving target which can always be improved.

1.8 Thesis structure

The thesis is structured as follows: The next chapter (chapter 2) provides a background description of the case study company, Topicus. The goal of the chapter is to give insight in the organization and its strategy. Topicus is a non-traditional development organization, the chapter explains how Topicus differs from other organizations. Chapter 3 is a literature study of scalability and software scalability. The goal of chapter 4 is to specify the draft model that is used for evaluation at Topicus. The draft model is created from existing and adopted evaluation models.

Chapter 5 is a summary of the research done during the project. It starts with the application of the proposed draft model and describes how it was tested. Aside from the practical adoption of the model, its impact on the organization is also evaluated. Chapter 6 discusses the key results of the research. It gives a structured overview of the findings and critically evaluates the lessons that can be learned. This chapter will also discuss the adoption of the roadmap in the organization. The organizational adoption is more than just introducing the roadmap. It has to be accepted and implemented by the organization. Last, in Chapter 7, a summary of the project results and the applicability is given. Next to this, recommendations are given for use, adoption and future research and extensions. The goal is to create a final overview of the project. References and appendices are placed at the end of the thesis.

2. Topicus

Topicus is a developer of web based solutions for within the Dutch marketplace [54]. What makes Topicus unique is their strategy for entering and growing in a specific market, and their organizational grow strategy. The beginning of this chapter explains the unique growth strategy of Topicus with respect to their competitors. The later sections describe the software development approach at Topicus.

2.1 Corporate strategy

The main difference between Topicus and other software developers is their strategy. Traditional developers approach the market from a single organization and try to cater different markets and customers. Topicus approaches the market differently. They use a 'multi-niche' strategy. Instead of being one big organization for multiple markets they create multiple organizations for separate markets [62].

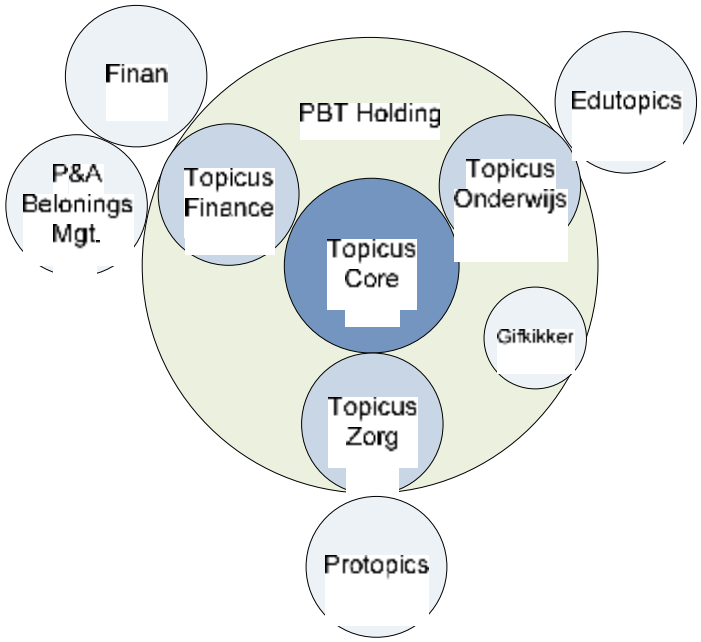


Figure 2.1: Organizational structure of Topicus

The advantage of this is that the separate organizations can specialize themselves in a specific market. This means that they can create a better fit between their organization and processes, and their target market. Topicus has three business units for separate markets. Topicus Finance for the financial sector, Topicus Onderwijs for the educational sector and Topicus Zorg for the healthcare sector. The organizations also branch off more specific independent or collaborative spin-offs.

Besides the development organizations with the name Topicus the holding has a number of non-development spin-offs and collaborative businesses for various non-development tasks. For example Protopics is a collaborative spin-off of Topicus, Promedico and PPMO (Participatiemaatschappij Oost-Nederland). The goal of Protopics is to be a dedicated maintenance oriented organisations to remove this burden from the other parties. This enables the original organisations to focus on their original mission. Another kind of spin-off is Topics' own special brew of beer called Gifkikker.

While the specific products are different for each market the overall visions of Topicus can be found in all their products. Topicus has an overall vision that is built on three important pillars:

- Information integration;
- Software as a service (SAAS);
- Organizational structure and process improvement;

The rise of the Internet allows communication between organizations that were traditionally isolated. Organizations that in the past could not share information, because of organizational and technological limitations, are now able to exchange knowledge. Connecting companies enables Topicus to expand their products and services throughout the sector. Building a good reputation is a key aspect for acquiring new customers.

Topicus develops products based on the "*software as a service*" (SAAS) concept. In the past SAAS was known as Application Service Provider (ASP). SAAS offers the use of programs as a service, where customers pay to use the program, not to own it. This concept is implemented with the use of different web applications which in turn are used to link different parties and users, who do not have to share a location.

This new way for sharing information and working independent of location has an impact on the organizational structure, and can be used to optimize processes. Activities that in the past would take too long, because of complexity and non-efficient communication processes, can now be linked together by integrating parts of the chain with web applications. The results of this strategy are encouraging. Topicus was at the top of the fastest growing business in the Dutch marketplace for the last couple of years. Topicus won the Deloitte fast 50 award in 2004, and has been part of the list in every consecutive year. It placed 21th in the Deloitte Fast 50 2007 award list with a 1028% growth.

2.2 Software development at Topicus

Topicus uses the Agile software development principles throughout the organization [6]. Agile software development is not a single strict method but a conceptual framework. Many different methods and approaches claim to be Agile, some of the best known are:

- Scrum
- eXtreme Programming (XP)
- Adaptive software development (ASD)
- Agile modelling (AM)

Topicus has adopted an Agile software development method because of Agile's focus on customer collaboration and it is based on best-practices of software development. Boehm and Turner wrote the book "*Balancing Agility and Discipline: A Guide for the Perplexed*"[6] describing both Agile and traditional development method and their respective strengths and weaknesses. They identified five characteristics that they call the *home ground* of Agile development. These five factors are:

- Low criticality of the application
- Senior developers
- Requirements change very often
- Small development teams
- A culture that thrives on chaos

Almost all of these characteristics can be found within Topicus. Topicus uses development teams of five to eleven members. Employees have a high educational level (all have a bachelor or master degree). This enables junior employees to gain a lot of experience and grow to a senior position in a couple of years. Using small development teams enables short communication channels and reduces management overhead. Each team usually consists of a project manager, one or two business analysts and between three to eight developers ranging from junior to senior level. The small teams are set up to work in close collaboration with customer.

This means during development requirements are subject to change. Changing requirements means the corporate culture has to be flexible, but flexibility breeds chaos. The low criticality is a point of interest. While the current products are already considered important, future projects will become more complex and critical. Therefore during future projects there is a higher need for quality assurance. The main reason Topicus is able to work efficiently with the agile programming method is that the company was designed to incorporate this method from the start. Other organizations often have great difficulty changing to an Agile methodology because their organization, culture and employees are stuck in the traditional development methodologies.

2.3 The FORCE Framework

Topicus Finance uses a component based framework called FORCE to assist and simplify the development of their software. The framework consists of different components which together are all potential parts of a system for the financial sector [16]. The FORCE framework was designed in 2005. It was a joint effort between Topicus and an independent financial service provider. Topicus and this financial service provider wanted to combine their experience and knowledge of the financial market and web applications into an ASP based platform which provides optimal support for both the financial intermediary and the financial product supplier. For the development on the ASP platform different essential components were identified which would form the FORCE framework.

The components are built around the six phases an intermediary goes through from acquisition of customers to the acceptation of a product and management of the contracts. The six stages are:

1. Acquisition
2. Advice
3. Proposition/sale
4. Administration/File creation
5. Financial administration
6. Contract management

During these stages several steps and connections have to be supported by the software. The components are divided into three categories: primary process components, support components and monitoring- and control components (see Figure 2.2).

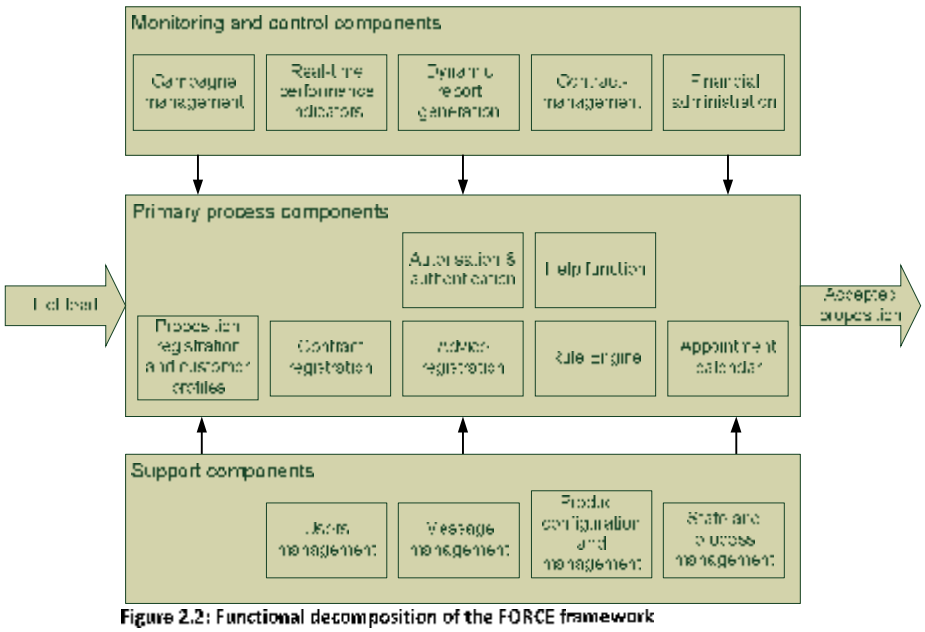


Figure 2.2: Functional decomposition of the FORCE framework

The different components are at the core of the Topicus reusable software library. The advantage of using the framework is having implemented and tested code for use throughout multiple projects. New projects and customers are used to explore and enhance the different possibilities of the framework. One of the growing concerns for Topicus is the ability of their applications to handle a growing user base over a larger amount of time. This is why scalability is an important subject for Topicus.

3. Exploring scalability: a literature review

This chapter forms the theoretical foundation for the project research. The goal is to provide an overview of the current state-of-the-art of scalability research. Scalability is a multidisciplinary research area disciplines like web engineering, requirements engineering, architecture design and non-functional requirements each influence scalability research. Previous work is used to better define the problem area and provide the foundation for the evaluation of scalability. The first section discusses scalability and the overall scalability issue. The second section discusses software scalability in greater detail. The last sections talk about the application, the differences and benefit of software scalability.

3.1 Scalability

Scalability is a critical, but very difficult non-functional requirement for a web application to predict and measure. This means it is an important subject for web application developers like Topicus. The ability to facilitate a large and growing number of users is of great importance in a competitive market. As future projects become bigger and more complex scalability will have a greater focus. The next section describes scalability in literature in greater detail.

3.1.1 Defining scalability

The word scalability is used throughout IT literature. Terms as size scalability, software system scalability, hardware scalability, application scalability, technology scalability, generation (time) scalability, space scalability, heterogeneity scalability, database scalability, interface scalability and more were encountered during the literature survey [24]. All these different definitions only make it more difficult to talk about scalability. Current research focuses on this definition problem and wants to bring structure to the scalability subject [15].

In scientific literature scalability is handled inconsistently and defined differently among authors. The different views can often be linked to the different environment of the author. Weinstock and Goodenough define scalability as [58]:

> *"1. Scalability is the ability to handle increased workload (without adding resources to a system)*
> *2. Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity"*

Bondi describes scalability as [8]:

> *"Scalability connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement"*

Williams and Smith describe scalability differently [60]:

> *"Scalability is a measure of an application system's ability to—without modification—cost-effectively provide increased throughput, reduced response time and/or support more users when hardware resources are added"*

And Duboc et al. describe scalability as [15]:

> *"We define scalability as a quality of software systems characterized by the casual impact that scaling aspects of the system environments and design have on certain measured system qualities as these aspects are varied over expected operational ranges"*

All these different views are typical for the way scalability is handled throughout literature. All definitions seem roughly the same, but differ when read in more detail. Common is the desire to increase the ability to handle a greater workload/number of users. Most literature deals with the issue of scalability from a hardware perspective, and define scalability accordingly. This hardware oriented view on the measurement of scalability is often described in terms of the gain when changing the number of CPUs or available memory [33], [60]. This research is interested in the scalability of the software of a system.

In the preliminary research scalability was defined as:

"Scalability is the ease with which a system can be expanded to serve more users and/or work"

The current level of scalability is often expressed in terms of *throughput*. Throughput is measured in the number of *user transactions per second* (tps). In this case a transaction is a user request from the browser to the server. The total number of transactions an application can handle is used to indicate the maximum load/users a system can handle. In that situation the application is scalable to that maximum load/users.

In 1990 M.D. Hill wrote a critical article named *"What is scalability?"* about the use of the term scalability for software systems [26]. Recognizing that without a good definition the term scalability has no real meaning, Hill called on the technical community to either rigorously define scalability or stop using it. At the time of writing this thesis however, 17 years later, there is still no real unifying definition of scalability. An ongoing PhD project by L. Duboc is focused on creating a comprehensive framework that encompasses the entire scalability subject. Her goal is to contribute to the reasoning and application, understanding and optimization of scalability. But at the time of writing this thesis her research is still in progress which means it is not known if this new definition will be supported by the research community.

With the rise of the Internet, web applications and web services, scalability has become a more important issue. Big companies as Ebay and Google who serve millions of users a day both require their services to be highly scalable. Google found it be to so important it organized a one day convention about scalability in July 2007 [23].

Current literature about scalability for software systems can be characterized as followed:

- *There is no clear definition of scalability. Many of the authors give a view on scalability related to their environment or desired goal.*
- *Scalability is often defined and analyzed in a specific context (like parallel computing or for specific application as video streaming).*
- *Scalability is focused on hardware (for example in networks or distribution among different servers and processors).*
- *Scalability for web applications is often described for large scale publicly accessible sites/applications as Ebay/Google. The focus is on the Business-to-Consumer(B2C) market.*

In the end a more unified definition of scalability is needed to bring the scalability research to the next level. The goal of this thesis is to further explore a small part of this issue.

3.1.2 The scalability problem

The reason scalability is interpreted differently by so many different authors is because of the inherent complexity of scalability. The scalability of a system is influenced by a lot of different forces and on a lot of different levels, of both hardware and software. In case of a web application there are many points between the server hosting a site or application, and the computer running the web browser, that impact the scalability. In Figure 3.1 every line and box has an impact on the scalability of the application.

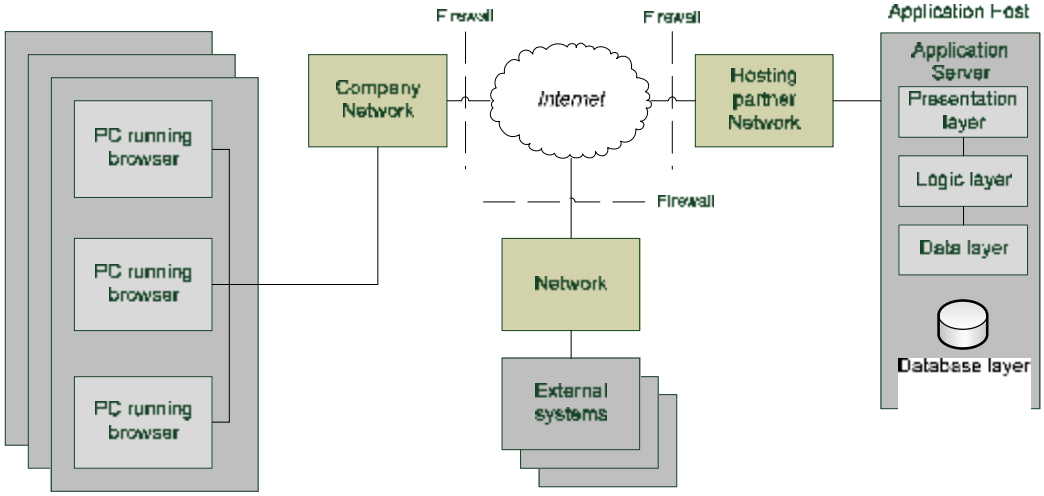


Figure 3.1: End-to-end overview of a web application

On the application server alone there are many points that, if implemented incorrectly, can cause scalability issues. In a *N*-tier architecture the scalability of the application can be evaluated as a complete entity, but also on the different levels. A bottleneck in the logic layer can cause the entire application to perform badly. This makes evaluating the scalability on a software level not as straightforward as evaluating the hardware scalability. When evaluating scalability it is always important to keep the bigger picture in mind, because scalability is a multi-faceted concept. Parts like the OS, the network and Internet connection speed and the programming language all influence scalability.

The ability to increase scalability is divided into two main approaches [60]. *Scaling up (vertical scaling)* and *scaling out (horizontal scaling)*. Scaling up is the improvement of hardware of a single server to improve the load capacity. Scaling out is the addition of servers to spread the load among multiple servers. The limitation of increasing scalability through hardware is that the scalability is not doubled just because a second server is added. Software is often not designed to make efficient use of the more powerful hardware or additional services.

Most organizations choose to improve the scalability of existing applications by adding or improving hardware. Hardware revision is considered to be less resource consuming than revisiting the code or building a new application from scratch. There is however a practical limit to hardware scaling. Adding servers also adds overhead for controlling all the servers. At a certain point efficient control of a large number of servers is a most impossible. Inefficient control leads to decreased scalability gain per added server. Vertical scaling is limited by the capabilities of the hardware components of the single server.

A more expensive, but on the long run possibly more efficient way is preparing the software for scalability. Only after the software is as optimized as possible, or the cost for further improvements is deemed too high, an organization should consider hardware scaling. A pro-active organization will use all the different levels on current and future applications to achieve the highest possible scalability an organization is willing to invest in. An organization should also apply lessons learned during the design stage of future projects to improve scalability of new implementations.

Another important aspect of scalability is that it is never perfect or finished. It can only be deemed acceptable for the current situation and expected growth. In essence scalability is a weakest-link problem. The part of the chain that has the biggest negative influence on the scalability of the application is the weakest-link. When that link is removed or optimized there will always be another part of the chain that becomes the *"most negative influence"* on the chain. As a result there are always (potential) bottlenecks to be found, in both the hardware and software. Once again the question becomes how much an organization is willing to invest in removing these bottlenecks. What makes the issue even more complex is the fact that scalability is not a fixed target. Acceptable scalability is not a static level. An application can never have too much scalability, only enough. This makes scalability a moving target issue.

3.2 Software scalability

The ability to improve scalability through optimization of code is often neglected. The few articles that talk about the scalability of software often explicitly link it to specific hardware configuration like in parallel programming. This research is interested in improving scalability without considering the hardware aspects. While software scalability can be improved or diminished with certain hardware components there are also practices and patterns that improve or hinder scalability independent of the used hardware. For example minimizing the exchange of data between the separate parts of the application. This is helpful because it decreases the transaction time and storage cost in every situation. Or using quick sort, which has an running time of O (n log n), instead of insertion sort, which has a running time of O (n*n).

Software scalability is creating scalability by creating scalable code. To this end a new definition is given:

"Software scalability is the ease with which the software of a system can be expanded to serve more users and/or work"

On a high level a software solution has to increase the scalability but also be reusable multiple time. Weinstock and Goodenough also recognized these two separate characteristics of scalability on a abstract level during their literature study and stated them as follows [58]:

1. Scalability is the ability to handle increased workload (without adding resources to a system)
2. Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity

This research attempts to give a more defined and practical approach to software scalability. The improvement and the applicability of the solution is expressed along two dimensions. A good software scalability solution has increase scalability and be reusable. This is very important because one of the defining characteristics of software scalability is the reusability of the solution. To this end software scalability is divided into two defining dimensions:

1. ***Software scalability is the ability to handle increased workload by changing parts of the code***
2. ***Software scalability has the ability to be used multiple times in a cost-effective way***

The first dimension describes the ability of a system to accommodate more users without a change in resource demand. How can the highest possible throughput be realized without using hardware specific abilities? This is described as the *scalability optimization* of a system. The second dimension is the ability to extend the scalability of a system with a cost-effective and repeatable strategy. Scalable software should not only work well on a single server, but it should also be portable to a distributed server setup or multiple projects. This is described as the *environmental flexibility*. When looking at software scalability both dimensions should be considered. And while the two dimensions are not mutually exclusive, it will be very hard to maximize optimization along both dimensions at the same time. A practical approach to software scalability is a trade-off between the two dimensions. Different alternatives will be either more or oriented to optimization, while other will be more flexible (see Figure 3.2). The terms *software optimization* and *environmental flexibility* are used to explain the two identified dimensions of software scalability.

These two parts of the definition can be linked to the idea of scaling up (vertical scaling) and scaling out (horizontal scaling) from the last section. Scalability optimization is the increase of scalability in an existing system through replacement. This can be seen as vertical scaling. The ability to extend the solution to multiple environments can be seen as horizontal scaling.

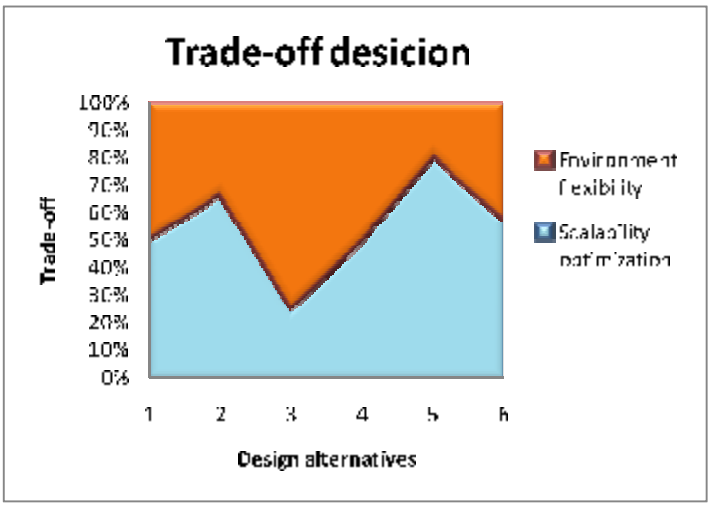


Figure 3.2: Trade-off decision between scalability optimization and environmental flexibility

3.3 Differences between hardware and software scalability

Scalability is an issue that spans both hardware and software. Different situations have different solutions. Some organizations have better opportunities for software revision where others prefer hardware upgrades. As previously mentioned, most organizations choose a hardware approach because it is an approach that is fast and simple. Software revision is often believed to be difficult and costly.

Hardware issues can be solved by replacing a part with a newer model that has a higher performance, for example replacing a CPU that is not fast enough with a faster version. A different approach is adding hardware resources to improve scalability, like adding wireless internet access points for better Internet coverage which improves the performance of the Internet connection. On the software side there is a larger distinction between different environments. The software environment is not just the application itself but also the operating system (OS), drivers and the database. Scalability problems in the environment are often out of the developer's control. Limitations in an OS like Windows or Linux cannot be removed directly but often need a workaround.

Almost every business application works with a database. And while databases are also software they are not built from scratch or mandatory for every application. This means the software developer has a greater area of control over a database. While a database and an application have an overlapping field they are considered to be different specialties. Database bottlenecks require specialized expertise different from general software knowledge. Databases can be implemented in different ways, such as relational and dimensional, each more suitable for a specific usage scenario or environment. Optimization and customization of a database is only possible within the chosen structure.

The last area is the code of the developed application. An error made during design and implementation can create scalability bottlenecks. The advantage of assessing part of the code is that a software developer has the required knowledge for making changes to the code, and complete control over the code. This makes bottlenecks found in this area are within the developers area of control and can be resolved easier.

An overview of the strengths and weaknesses of the different bottlenecks give an indication which areas are more suitable for change (* = depends if the source code is open-source Table 3.1). This overview is made from the perspective of a software developer. This overview shows the potential for improvement of scalability by changing parts of the application code.

Bottleneck area	Changeability	Knowledge required	Ownership of code
Hardware	High	Low	Yes
Software Environment	Low	High	No*
Application code	High	High	Yes
Database	Medium	High	Yes

* = depends if the source code is open-source

Table 3.1: Scalability bottleneck categories

There are advantages for choosing software over of hardware scalability. In contrast to hardware, software optimization is usable among multiple applications. Adding a server for a single application does nothing for other applications, while optimizing a component that is used in multiple projects can benefit all of them. Knowledge gained from using different solutions can be used reactively and proactively in the future. The knowledge can be used to optimize the application during the design which has a greater impact on scalability. This makes investing in software scalability a valid option for any organization. Organizations that use a lot of servers (like Ebay/Google) or a minimum amount of servers (like Topicus), the gain from software optimization is particularly interesting.

A disadvantage is the potential complexity and resource demand of changing code. Assessing and optimizing the scalability of a complex application is not an easy undertaking. The more complex the environment and the application, the more difficult it will be to optimize the software. This also means that the cost for improving scalability per server is high, especially if it is only calculated for a single or small amount of servers. If the code and/or knowledge is reusable, then the cost per server decreases. The other disadvantage is that the maximum load increase is typically larger when adding a server then when optimizing the software using the same amount of resources. A small overview of the different pro- and cons is given in Table 3.2.

Bottleneck area	Changeability	Initial costs	Reusable	Knowledge generation	Complexity
Hardware	High	Low	No	Low	Low
Software	High	High	Yes	High	High

Table 3.2: Comparison of hardware-software scalability investment

Testing software scalability also differs from hardware scalability in the information gained from the testing. If a specific hardware component is delivering unacceptable performance a simple solution is replacing the component with a faster and better one. If a performance issue is identified in the software the answer is not that straightforward. The limit with testing is that while it is usable for finding issues it does not give any real information about the cause of the issue.

3.4 Investing in software scalability

The economic side of a scalability solution is an important part for choosing an approach. When evaluating different options for the associated cost are an important factor for most organizations [60]. The complexity of decisions about scalability is that its payoff lies in the future, and future scalability is hard to predict. Some organizations will choose for major investments in scalability, for example redesign of the entire infrastructure, while others will choose for a more gradual approach, adding one server at a time. The most cost-effective solution at the present may not be the best solution for the future. Improving an application along the two dimensions might not be cost effective in short term, but it can be a worthwhile long term investment

Optimizing for scalability during design increases improves the chances for scalability in the later stages. An application that is built with scalability as in important characteristic in mind improves scalability of the implementation. As with many issues, problems are best handled at the core (eg. earliest possible opportunity). The earlier scalability is "implemented" the bigger the impact on the outcome. The impact on the different levels of optimization for scalability can be shown in a pyramid shape (Figure 3.3).

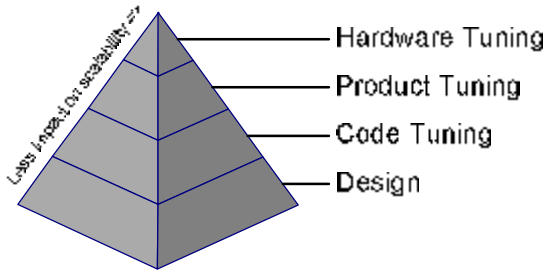


Figure 3.3: Scalability pyramid

Like bug fixing the earlier the scalability is taken into account the lower the cost of fixing scalability issues [47]. Implementing scalability during design is cheaper then revising code during operation (Figure 3.4).

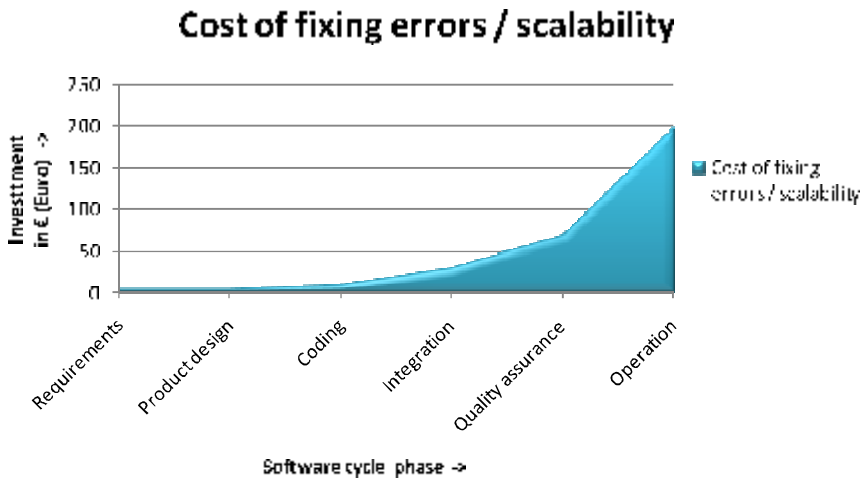


Figure 3.4: Cost of bug fixing per development cycle

Assume that a given investment in software scalability increases the load capacity with 5%. When an organization uses one server, with a normal capacity of 100 concurrent users, can now handle 105 users, so the need for a second server is delayed. And when the second server is used, and the additional overhead penalty is assumed to be minimal, both of them can handle 210 users. Instead of the 200 if the original server was duplicated. In June 2006 the New York Times estimated that Google has at least four hundred fifty thousand (450.000) servers to be able to offer all of its services [34]. If they could increase the scalability of their software with 5% they would only need 428.572 servers to handle the same load while being able to sustain the same load the organization could remove 21.428 servers. This would also reduces the maintenance effort and costs and the energy consumption.

These savings may also be important for the growing interest in green computing [56]. Across all industries and borders there is a growing demand for energy efficiency in both the home and work environment. More effective use of the available hardware is one of the goals of this trend. And in a time were environmental friendly energy consumption is becoming and hot issue the ability to handle a larger load per energy consuming server is an advantage.

Software optimization is not recommendable in every situation. An organization should always check which approach is the most rewarding. A situation in which an organization can reuse the software solution multiple times increases the chance of a high return of investment. A reusable software solution has a high chance of generating a bigger return of investment than a hardware solution. A hardware investment will remain the same for every iteration, while reusing high quality code becomes more efficient (see Figure 3.5).

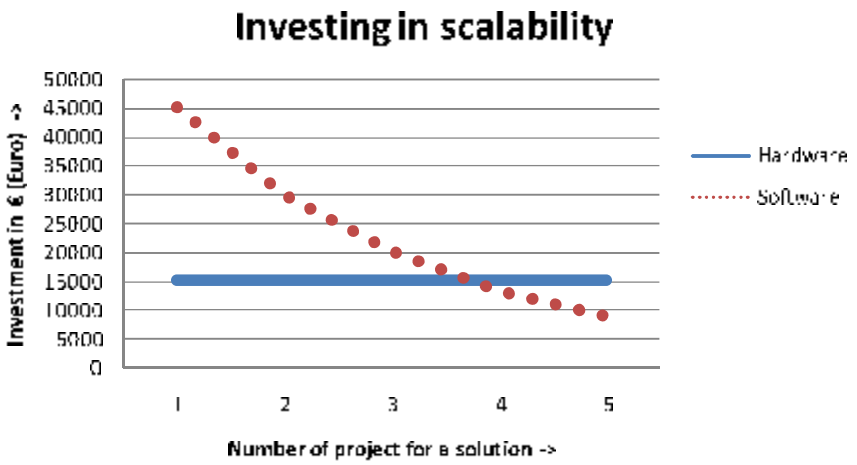


Figure 3.5: Hardware versus Software investment

The large scale and complexity of scalability make it hard to approach scalability without a structured approach. To simplify the scalability environment this research approaches scalability from a software perspective . To measure the scalability of software and identify potential bottlenecks an evaluation method for software scalability is needed. The next section gives an overview of currently available evaluation methods.

3.5 Current software scalability evaluation methods

Evaluation is the assessment or statement of value [9]. This assessment and generation of explicit information about software scalability is necessary for the further improvement of software scalability. This section is about current methods that may be applicable for the evaluation of scalability

One of the most common approaches for the evaluation of software is through the use of metrics. The existing quality models of Boehm, McCall and ISO 9126 propose different metrics for the evaluation of non-functional requirements. None of these models were designed for the evaluation of web applications and none of them use scalability. Almost all metrics in scalability literature revolve around the measurement of throughput in specific scenarios [28], [31]. Specific metrics exist for parallel computing [31] and distributed computing [28]. In these scenarios the hardware is often the most important factor for the change in scalability. This thesis is interested in the scalability of software on which hardware is not a direct influence. To this end a few available scalability evaluation methods are compared.

Stress/Load testing

Perhaps the most applied evaluation method for scalability is stress and load testing [19], [53]. However, testing to see if an application adheres to the functional demands, is only part of the entire testing process. Another part is testing various non-functional aspects such as performance, security, scalability and usability. Testing scalability is done to analyze the scalability of a system. Scalability testing is often divided in load and stress testing. Load testing is testing the system against an expected load for the system. The goal is to identify the optimum number of users for the application while the performance requirements are still met.

Stress testing is testing an application under stress conditions. Where load testing is used for finding the currently achievable optimum, stress testing is about pushing the application beyond this point. The goal of the stress test is to find the maximum load a system can handle before it becomes unusable. The objective of this kind of testing is to quantify scalability and identify (potential) scalability problems. Knowing what the current scalability limit is gives the developer the opportunity to compare it to current and future projects.

The Probability of Non-scalability Likelihood (PNL) metric

One of the few scalability metrics that is not linked to hardware is the Probability of Non-scalability Likelihood (PNL) metric of Weyuker and Avritzer [59]. The PNL metric is used to predict whether or not the software system will be able to handle significant more load than it currently does. The metric is calculated with the formula

$$PNL(P,Q) = \sum \Pr(s)\, C(s)$$

Pr(s) is the probability of a given state s and C(s) is the acceptability of a certain performance, C(s) is 0 when the performance is acceptable and 1 when it is unacceptable. The problem with using the PNL metric is that in order to make a valid prediction a huge amount of field data is required. Gathering this data for a project of reasonable size can take up to several months. This makes the PNL metric difficult to use in practice. The authors themselves propose daily extensive data collections to gather enough information.

The Quantitative Scalability Evaluation Method (QSEM)

Besides the methods that directly measure or compare scalability other approaches give an evaluation framework for scalability. One of these frameworks is the Quantitative Scalability Evaluation Method (QSEM) by Williams and Smith [61]. QSEM evaluates measurements to quantify the scalability capacity of a system. They define scalability as a system property, encompassing both hardware and software. The software architecture and the execution environment are identified as key factors for achieving scalability. QSEM provides a short roadmap for thinking about, preparation of and measurement of scalability. QSEM evaluates scalability with the help of seven steps:

1. Identify critical Use Cases: Identify the externally visible behaviour of the software that are critical to responsiveness or scalability
2. Select representative scalability scenarios: For each critical Use Case, identify the scenarios that are important to scalability
3. Determine scalability requirements: Identify precise, quantitative, measurable scalability requirements
4. Plan measurement studies: Identify the bottleneck resource, plan measurements, develop load generator scripts, determine what parameters to measure, identify measurement tools, and document the test plans
5. Perform measurements: Conduct the measurement experiments, collect data, and document the results
6. Evaluate data: Evaluate the measurement data to determine whether the scalability requirements can be met and select the best scaling strategy
7. Present results: Present results and recommendations to stakeholders

The authors of QSEM define scalability as a system property, their examples focus on evaluation of different execution environments. The focus of this thesis is on optimizing the software for scalability. It is not an evaluation method for different alternative environments. While the two share certain similarities, the goal for each evaluation method is different. An evaluation model for alternatives is used as a decision-support method where an evaluation model for optimization is used to identify (potential) problems and structuring solution approaches. Measurement only indicates a problem but does not give specific information about possible causes and solutions to this problem.

Mishra's scalability comparison method

Mishra approaches scalability from a comparison perspective [39]. The paper proposes a method for evaluating the scalability of a changed system given a specific performance objective. The method uses a couple of equations that enable the comparison between the existing and the proposed system. If the outcome is acceptable the changes can be implemented. When the performance objectives are not met the first suggestion is to make changes in the hardware architecture. In the case changes in the hardware do not produce the required results, changes in the software architecture should be studied. Only when, even with the changes, the desired requirements are still not met should a user revise the earlier stated requirements. Like QSEM this method is aimed at evaluating the usefulness of an alternative and not for identifying and solving scalability bottlenecks.

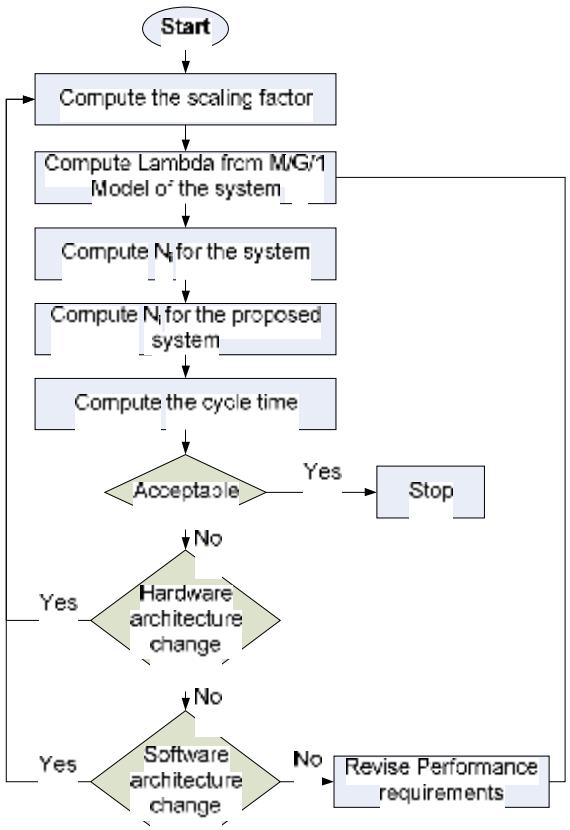


Figure 3.6: Mishra's scalability comparison flow chart [39]

Duboc et al. software system scalability framework

A framework that is more tailored to scalability is the software system scalability framework of Duboc et al. [15]. The framework is designed to reveal the underlying causal relationship between certain *factors* and certain *dependent variables*. The factors are divided along two dimensions. The factors are identified as either scaling or non-scaling. The other dimension is the source of the factors, it being from the machine domain (hardware) or the application domain (software). The factors that can be manipulated during the scalability analysis are called independent variables. The factors that are static and cannot be manipulated are called nuisance variables. Factors that change when independent variables are manipulated are dependant factors.

A scalability analysis is the analysis of a dependent variable in presence of a variation of the scalability dimensions. This means that one should not refer to the scalability of an entire system because this breeds ambiguities to what the "system" means in this context. A more precise approach would be the question what the "scalability with respect to the CPU user" or the "scalability of users with respect to the current disk space"

The framework is driven by the need of a stakeholder. The framework is question-based because the answer whether or not the scalability is acceptable comes from the results of the dependent variable analysis.

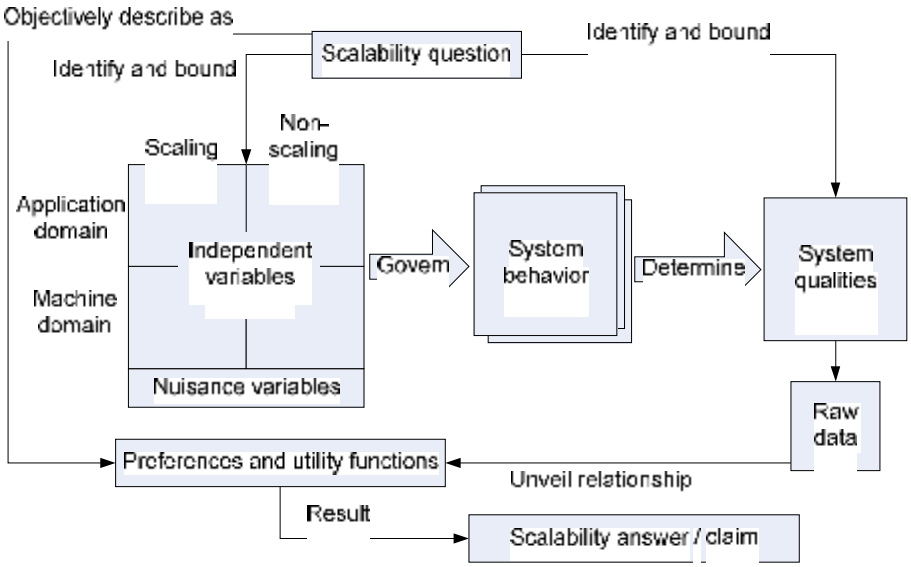


Figure 3.7: Scalability framework of Duboc et al [9]

Evaluation method overview

Just like the definition of scalability there is a broad range of evaluation methods for scalability. The models focus on the comparison of alternatives or giving a roadmap for the evaluation process. They are however not usable for identifying and solving actual scalability problems. This research is interested in finding and solving software scalability problems. To be able to achieve this, software bottlenecks have to be identified and analyzed. The current evaluation methods are not directly usable for this, so a new evaluation is needed.

4. Software scalability evaluation model

In this chapter, a new, more precise method for evaluating software scalability is proposed. This new method combines aspects from existing methods and introduces new concepts which are tailored to software scalability. The later sections explain the different tools and measurement than can be used in the new evaluation method and demonstrate them with the help of an example.

4.1 A new software scalability evaluation method

None of the previously mentioned evaluation methods are usable for measuring and evaluating software scalability. Let alone in a simple and effective way. The reason for this is the inherent complexity of scalability. Scalability is dependent on multiple criteria and is often linked to characteristics such as infrastructure, programming language and OS. Evaluating the scalability over that many characteristics and environments is virtually impossible.

Software scalability is focused on the executable code of an application. In the case of web applications this code is located on the application server. The scalability of the operating system, drivers and the database are not taken into account. Like any hardware these software parts are considered static and unchangeable in this context. The model for evaluating software scalability should be usable in different situations as well as usable for the given problem statement. The overall goals for the evaluation model are the following:

1. The model must be usable to evaluate the scalability of software
2. It must incorporate the two dimensions of software scalability, scalability optimization and environmental flexibility
3. It must guide the user in steps to identify potential scalability bottlenecks and solution approaches
4. It must promote awareness of scalability and scalability problems in the organization
5. It must be practical to use in a business environment

To evaluate software scalability for improvement, instead of comparison, a new and more tailored model is needed. This part of the thesis is primarily interested in finding and solving software scalability bottlenecks. The evaluation methods for scalability from the previous chapter focus on different goals. To this end a new model is proposed that fulfils the above mentioned goals.

The first step of exploring software scalability is evaluating the current scalability. Of interest are whether or not there is an actual scalability problem and what the potential bottlenecks are. The second step is finding the cause of the bottleneck(s). It is easier to solve a problem when the cause and effect of the problem are understood. The final step is applying some kind of change to the software that improves or removes the bottleneck. In summary the model guides the user in answering the three following questions about software scalability:

1. *What are the bottlenecks that limit the scalability of the application?*
   (eg. Which parts of the code influence software scalability)
2. *What is the cause and effect of the bottlenecks?*
   (eg. How are software scalability characteristics handled in the code)
3. *How can the bottleneck be removed and the scalability improved?*
   (eg. How should the software scalability characteristics be handled in the code)

Software scalability on a code level is not directly measurable. The two dimensions of software scalability, scalability optimization and environmental flexibility can be linked to other aspects in Software Engineering. Scalability optimization can be seen as code tuning. Environmental flexibility can be linked to concepts as effective implementation, modularity and maintainability of the code. Any concept that helps a developer to better understand and modify code is useful for environmental flexibility. By using code metrics to indirectly indicate scalability the current scalability can be evaluated. Combining the test data and the code metrics should give information about the overall scalability, where potential bottlenecks are located and what some of the more likely causes are. Guidelines for creating scalability within a certain programming environment should help with improving certain scalability performance bottlenecks. A visualization of this problem is given in Figure 4.1.

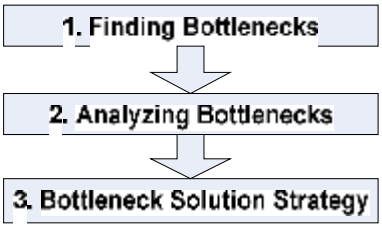


Figure 4.1: Global evaluation steps

4.2 The evaluation method explained

Each step has its own specific goal and applicable tools or methods. To explain these steps in greater detail example data is used. This data is only placeholder and not from actual tests. The data is used to give a comprehensive example of how the gathered data is usable in the scalability evaluation.

4.2.1 Finding software scalability bottlenecks

How can software scalability bottlenecks be identified? Bottlenecks can be found by using various methods. A good starting point is the performance and usage data of the web application. Searching is done by actively gathering information. Gathering information about scalability of an application can be done by testing. Testing for traditional software systems is well known and understood throughout Software Engineering [41]. The testing of web applications differs somewhat from the traditional methods, because more factors influence web applications (especially from an end-user perspective). For traditional systems the testing environment can be made as a reasonable simulation of the actual environment. Making the same reasonable simulation for a web environment with lots of different aspects is extremely difficult. To this end more tailored testing methods for web application are proposed [52].

Just as important as deciding which method and tool to use for the evaluation (*the how*), is deciding what information is desired (*the what*). The first couple of steps of the QSEM method represent a structure by which to gather and prepare a performance test. It is important to know which information is important and what it represents, for a good analysis of the application (*What data do you want need for a good analysis?*). The second part is preparing relevant applications and generating test scenario's (*How do you gather the data?*). And the last part is interpretation and analysis of data (*What does the acquired data mean?*).

Log files

Information about the current scalability of a application can be found in the load and performance data of the system. One of the two dimensions of software scalability is the throughput optimization. Knowing how long certain it takes a web application to execute certain use cases gives an indication of which parts influence performance. These are candidates for scalability bottlenecks. Usable information is the number of users of an application, the total number of pages requested, the time spent creating and sending these pages and the amount of data send.

This information can be generated on demand by using performance testing tools or from log files. Scalability testing tools are available for different applications and programming languages. They exist both as stand-alone commercial or open-source programs and as plug-in for development application as Visual Studio and Eclipse. Almost all of these test tools allow the user to program or record a set of activities of a web application which can be later used as a performance test. Load testing tools use these recordings to simulate multiple users executing these scenario's. During execution information is collected that is used to search for potential bottlenecks. Usually the same information can also be obtained from the log files of the application server. The advantage of the log files is that the information is not from one or a couple of trials and therefore is more realistic because of the greater number of tests. The drawback is that they cannot generate new information on demand and often also contain information the user might not be interested in. An example of potential information is given in Table 4.1.

URL	Avg. Page Time (sec)	Avg. Page size (byte)	Count
Example Page 1	0.02	328	21
Example Page 2	0.5	578	65
Example Page 3	1	110.347	11

Table 4.1: Example data from server of test logs

Stress and load testing

For scalability the most important aspect is the generated amount of work (throughput) under a certain condition, like the amount of users. Load testing is measuring an application under a specific workload. Evaluating load can be done by looking at the maximum number of users that can be serviced while checking if the performance does not drop below a certain performance threshold. One could also fix the number of users and measure the maximum workload that can is subsequently generated. The first strategy is best used when the workload itself is fixed and an organization wants to maximize the number of users per server. The second is useful if the scalability for variable workloads is more important than the number of users.

Scalability testing can also be done positively and negatively. Positive testing is checking to see if an application behaves as expected. Load testing is part of this category. The opposite is negative testing. Negative testing is seeing what it takes to break an application. Stress testing is trying to break the system. What does it take to make an application fail and how long does it take to recover?

During stress testing the performance of a single test is not very important but the overall performance of the server is. The web application is targeted by multiple instances of one or more scenario's and tries to handle them as fast as possible. The degree with which the server can handle concurrent users reflects the scalability of the system and its scalability bottlenecks. Important information from a load/stress test is the load the users are generating (request/sec) and the load of the server and any connected database.

Number of users	Request / Sec	Avg. Response Time (sec)	Server CPU load (%)	Database load (%)
10	22	3	40	20
20	44	5	65	30
40	76	15	88	50

Table 4.2: Example information from load or stress testing

The information from the scalability test can also be depicted visually to give a quick overview of the scalability of a system/module. The advantage of the visual representation is that it gives a simple overview of the result.

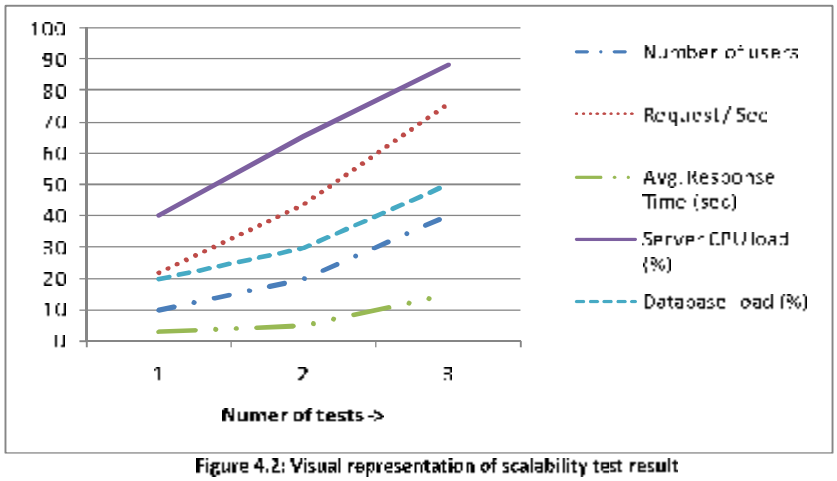


Figure 4.2: Visual representation of scalability test result

Bug tracking system

Bug tracking systems are also a source for bottleneck identification. Bug tracking systems allow the administration and tracking of issues. Inside the tracking system bugs are recorded along with information that may be needed for analyzing and solving the problem. Different information related to the bug can be tracked as well as the importance of the bug, where and when it was discovered and who is working on the issue. Like the log files the bug tracking database is full of possibilities for bottleneck identification. The obvious source of information are performance issues. Depending on whether performance requirements were stated during development these bugs range from critical to optional. The less obvious source is looking at the number of bugs certain parts of the application have. A high number of bugs can implicate a very complex or unstable part of the application. Both reasons can warrant a more detailed inspection of the application.

Expert Knowledge

The experience of the people involved with the application is also a source of information. Especially if the professionals are directly and actively involved in the design and implementation of the application, they possess a wealth of information. The big advantage is that this approach is easily implementable. IT professionals should be able to use their IT experience and reasoning skills to analyze the application. This analysis can be used not only to identify potential risks but also to reason about their solution. Creating an opportunity to discuss scalability and scalability issues, like a meeting or questionnaire, can generate understanding of the scalability of the application and identification of weak points in the implementation. Brainstorm sessions or asking the members of the development team to each make a top five of potential bottlenecks is not only useful for finding bottlenecks but also for encouraging the development team to think about the scalability of their applications.

Bottleneck Ranking

If multiple bottlenecks are found it is important to rank the bottlenecks on priority and predict the resources needed to fix the bottleneck. The ranking and resource prediction makes it easier to discuss the different alternatives and decide which bottleneck to approach first. While multiple bottlenecks can be tackled simultaneously, the removal of one bottleneck can affect others both positively and negatively. For simplicity this paper approaches each bottleneck individually and assumes the dependency between bottlenecks is minimal.

4.2.2 Analyzing software scalability bottlenecks

After bottlenecks are identified and ranked the next step is analyzing the cause and effect of each bottleneck. Different bottlenecks require different solutions. To analyze the cause and effect of a bottleneck the first distinction made is between the two dimensions for software scalability introduced earlier: *throughput optimization* and *environmental flexibility*.

Code Analysis

Scalability optimization is part of performance optimization of an application. Performance gain is defined in a broad sense, not only time but also memory and storage space are important. Time can be gained from a faster execution of a method but also from shorter communication lines. For example a proposition containing a name, address, customerID and 20 other variables is exchanged between a database and a method. The easy solution is the exchange the entire proposition without looking which information is actually needed. In case of a large record a lot of information needs to be retrieved and evaluated. A more efficient, but more complex, method is only selecting the appropriate information from the database.

A tool to analyze the performance demand of an application is a profiler. A profiler is a tool that measures the behaviour of an application during runtime. It can be used to measure CPU and memory load but also function call duration and count. A profiler goes further than a test tool and gives information about which part or call of an application takes up the most resources. Applications that have a high resource demand, like memory, leave less of that resource for other methods which has an influence on the performance. The profiler also gives insight into how a specific call is handled. Some programming rules can have unintended side effects that take up system resources. For example the wrong use of locks on threads in *C#* can degrade the performance severely.

Code Metrics

Environmental flexibility is not an existing quality attribute. Being flexible is the ability to easily change parts of application code to fit new needs and wishes. Flexible software requires no or only a minimal amount of change to take full advantage of changes in the environment. Characteristics as complexity, testability and modularity are part of software reuse [32]. The main goal of reusability is to provide code that is not only of high quality but also easily implementable throughout different environments. These goals can be linked to modularity, maintainability, reusability and portability. While scalability cannot be directly measured from code, scalability factors as modularity and maintainability are more established. A number of factors are expected to be important for scalable software:

- Coupling
- Cohesion
- Complexity
- Instability

Coupling and cohesion were first proposed by Stevens, Myers and Constantine in 1974 [51]. Coupling is the amount of interdependency between software parts. A module that depends on information from other modules is called dependent. Changes in one can affect the other. The dependency becomes even greater if the other modules are once again dependent on another module(or the first module). This way all kinds of dependencies between modules arise. A high amount of coupling result in complex and potentially unstable code because changes are hard to make and it is ever harder to predict the effect. To have flexible code that can be scaled to different environments, a low coupling is desirable. Cohesion is the degree of which a the part of a single module consist of closely related operations. A model needs a clear and apparent task and should is related to the function of the module. A high cohesion is linked to desirable flexibility traits as reusability, understandability and robustness.

Complexity can be expressed in different ways. Code complexity can be judged objectively, based on measurements, or subjectively, based on user perception. One of the best known complexity metrics is McCabe's cyclomatic complexity. In 1976 McCabe proposed a metric that measures the complexity based on linearly independent paths through the application [35]. The more paths there are, the more complex the code is: a higher value depicts a greater complexity.

Instability of a module is a ratio between 0 and 1 indicating the stability of your class. Zero means completely stable and one is unstable. Instability is calculated from the number of outgoing dependencies (efferent coupling(EC) and the number of incoming dependencies (afferent coupling(AC). The formula for this calculation is

$$Instability = EC \div (AC + EC)$$

A module becomes unstable if the majority of dependencies are outgoing. A majority of outgoing dependencies is, like the coupling metric, indicative of fragile code. If a change in one part has an effect on a great number of other connected module that code is considered to be unstable.

Depedency Structure Matrix and Graph

Another method to visually represent the dependency of an application is the use of Dependency Structure Matrices (DSM). DSM are also known as Problem Structure Matrix (PSM) or Design Structure Matrix (see Table 4.3). The matrix represents dependences between parts of an application. The matrix has horizontal and vertical dimensions that list methods or modules of an application. If a part depends on another part it is marked as a dependency either as a more traditional X or as a number representing the amount of dependencies. Dependencies are either direct or indirect. A direct dependency is if module A calls on module B. An example of indirect dependency is a module C that is directly dependent on module B, whereas module B is directly dependent on module A. In this case, module C is indirectly dependent on A.

Different information can be gained from analyzing and manipulating the matrix. First the number and spread of the dependencies gives insight into the complexity of the application. Analyzing the matrix gives more specific information about which part is most dependent on other modules. The matrix can be used to validate certain architectural decisions like using a (strictly) layered systems. In a layered system no dependency above the grey line may occur. In a strictly layered system a part may only access the part directly above it. The dependencies can also be visualized in a dependency graph (DSG). The graph shows the same information as the matrix but represents it visually (see Table 4.3 and Figure 4.3).

	Module A	Module B	Module C	Module D	Module E
Module A					
Module B	X				
Module C		X			
Module D	X				
Module E		X		X	

Table 4.3: DSM example matrix

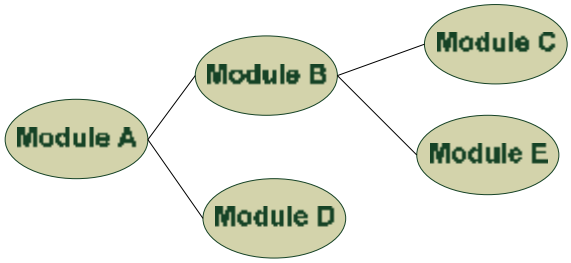


Figure 4.3: Dependency graph of Table 4.3

Software reusability

Besides metrics about the environmental flexibility, there are factors that give information about the reusability of code. The ability to reuse code is an important advantage of software scalability compared to hardware scalability. If parts of the code are modular, stable and simple there is high potential for reuse. The actual reuse of code can be measured by looking at ratio of reuse. The amount of commentary lines that help other developers in reusing the code is the documentation rate. For some basic insight into the reusability of code two metrics are proposed:

- Reuse of code
- Documentation rate

The reuse of code metric gives insight in which parts of the code are build for reuse among multiple projects. The documentation rate is an important factor in promoting reuse because information about the purpose and implementation of the code help developers to simplify and speed up code reuse.

The documentation rate is the percentage of lines dedicated to documentation compared to the total lines of code. A high percentage of documentation makes reuse easier because functions and modules are clearly documented. Code that is sufficiently documented is easier to rewrite or reuse because there is extra information on what the code does. The optimal amount of documentation differs per application. A low document rate has a high chance of explaining to little, but a high documentation rate has a chance of explaining to much. Both decrease the ease with which code can be reused. One of the tools used to measure the documentation rate recommends a rate between 20 % and 40% [42]. The problem with the documentation rate is that it does not measure the quality of the documentation, it only looks at quantity. While the metric is a good global measurement the actual usefulness remains unclear. Clear documentation guidelines can improve the documentation quality. Reuse of code is an important when code needs to be considered flexible. Code that has to be rewritten extensively to accommodate a changed environment is not considered flexible.

In the end there are a lot of possible measurements and tests that can be done to analyze the problem. This thesis does not propose to be complete or have only the best options. The metrics and measurements chosen are only a small fraction of all possible options. The current proposed methods were chosen because they represent well understood and well defined values, or because of applicability or interest. Other methods and measurements that provide insight in to a bottleneck are just as applicable. It does not matter how you gain understanding of the problem, just that you gain enough of it, to solve the problem.

4.2.3 Software scalability bottleneck solution strategy

When the bottleneck is understood the final step is to use the knowledge about the bottleneck to solve the problem. The main problem with all the different scalability issues is that the solution has to be tailored to the situation. The specific solution has to take the environment, the infrastructure, the programming language and the architecture into account. A more preferable approach is searching for performance and flexibility solutions related to a specific aspect of the application. This knowledge can be gained from whitepapers, community forums and best practices. People who work extensively with a certain technology gain a useful insight into the proper application of the technology.

Topicus Finance develops web applications based on the .NET framework developed by Microsoft. Microsoft wants to help developers getting the most out of the creating white papers with insight into their language. They published white papers on "Performance Testing Guidance for Web Applications" and "Improving .NET Application Performance and Scalability" to help developers improve their .NET applications [37], [38]. Additional insight is available through community sites and discussion boards. Sun has similar guides for Java development in both books and online resources [5], [10].

To help create scalability during design a number of "*scalable*" design may be useful. Ahluwalia proposed a series of ten patterns that describe different design choices for creating a scalable application [2]. Some of these talk about software characteristics, such as algorithm optimization and other about hardware and hardware related, like adding parallel programming and the associated hardware requirements.

4.2.4 Knowledge and software reuse

Just as important as fixing the actual bottlenecks is the *reusability* of the new code and of the acquired knowledge. Both aspects are done when bottlenecks are solved but each is usable in a different way. The concept software reuse was introduced in 1968 [18], [30], [50]. It entails the reuse of existing software in new situations or applications instead of rebuilding it from scratch. Reusing existing software has the advantage that it decreases implementation time and improves quality because it was already examined after its initial implementation. While many organizations strive for efficient and effective reuse, a lot off organizations struggle with its implementation [17], [18]. Knowledge reuse is a more abstract approach to the reuse of information [27], whereas software reuse is a more explicit and codified type of knowledge.

Knowledge is roughly divided into two sides, *tacit knowledge* and *explicit/implicit knowledge* [43]. Explicit/implicit knowledge is knowledge that a person is aware of and which can be easily transferred to other persons or recorded. The knowledge is implicit when it can be recorded or transferred and explicit when it already has been recorded or transferred. Recorded best practices, manuals and online resources are examples of explicit knowledge. Explicit knowledge often describes information in facts and methods.

Tacit knowledge is knowledge that a person is not actively aware of and is often regarded intuitive, therefore this kind if knowledge is not easily recorded. Tacit knowledge is more procedural in that it is a representation of what is being done. Transferring tacit knowledge is usually done by observation and imitation instead of declared forms of knowledge. This makes tacit knowledge a lot harder to share and communicate which poses a problem for knowledge reuse. An overview of these concepts is given in Figure 4.4.

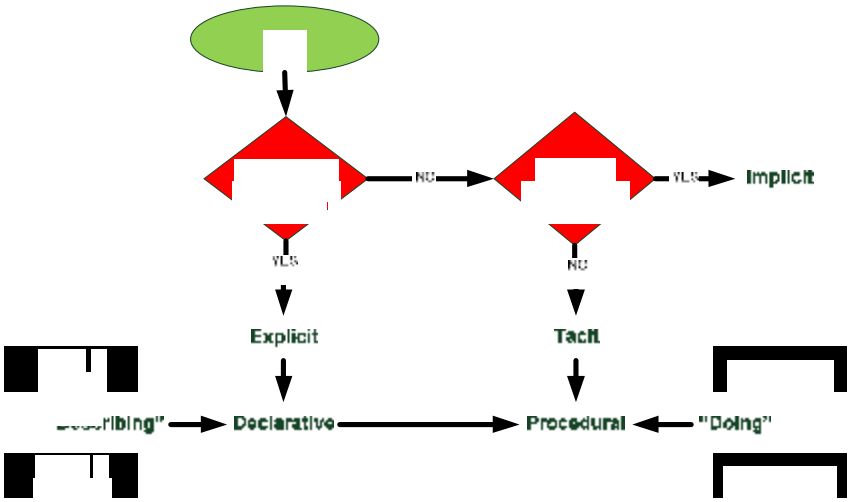


Figure 4.4: Knowledge management concepts from [48]

Different authors recognize the importance of tacit knowledge in creating innovation [25], [49]. The improvement of software scalability requires the introduction of new and changed parts of code. Researching and trying new ways to improve scalability can be seen as innovation and this makes tacit knowledge an important part of scalability. Improvement can be achieved by producing reusable software components and best practices (explicit knowledge) but also by generating awareness of the scalability issue (tacit knowledge).

Ideally most of the scalability know edge will have to be made explicit for easy reuse. Explicit know edge is easier to share amongst the organization than tacit knowledge. It will however be difficult to codify all knowledge from tacit to explicit. Writ ng best practices, guidelines and organ z ng workshops will help an organization in sharing information about sca ability.

App ying these steps in multiple iterations can ncrease the effect of knowledge generation. Each cycle identifies another bottleneck and each individua solution adds to the knowledge base and quality of the software.

4.2.5 Evaluation model overview

Adding the different tools and methods to global evaluation steps (Figure 4.1) now gives a more complete approach for analysis of software scalability (see Figure 4.5). In practice different methods and tools can be inserted in a specific step. As mentioned earlier, how the insights is gained is of lesser importance. The objective is to gain usable and reliable information for finding, analyzing and solving scalability bottlenecks.

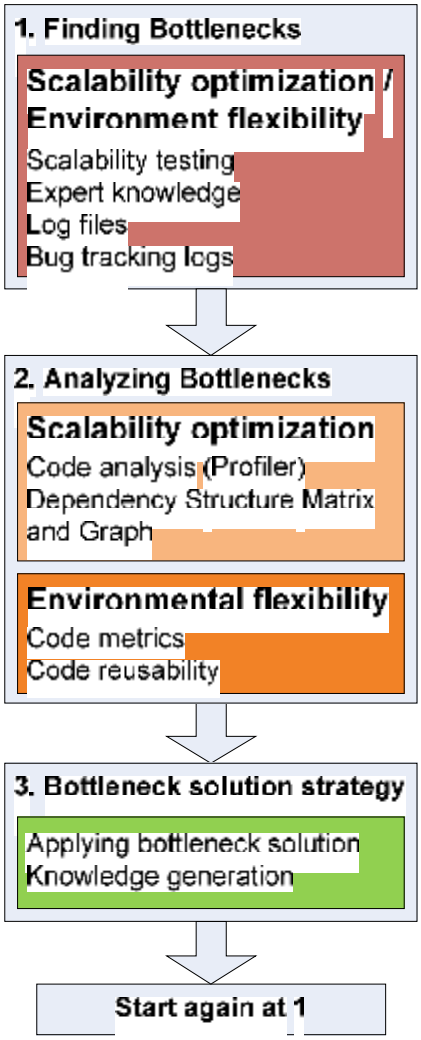


Figure 4.5: Using different software scalability evaluation tools/methods in the three evaluation steps

5. Topicus case study

The following chapter is a case study of the scalability at Topicus. As mentioned in chapter 2, Topicus is a development organization based on the agile developments method. For the financial market they have a set of modules that represent stakeholder demands. In the next section the three evaluation model questions will be answered using a real-life Topicus application. This chapter gives the highlight of the results and an interpretation. The full unedited results are available in Appendix B.

5.1 Case study settings

The application under inspection is a mortgage broker system, henceforth called application A. The system is used by intermediaries to sell and register mortgages. On the mortgage provider side it is used for different administrative functions. The main use case is the registration and acceptance of a new mortgage product. The application is built in the .NET framework and uses different languages and methods like C#, AJAX and Javascript. In the past this application has not been specifically optimized for scalability. This means it should still have opportunities for bottleneck identification, analysis and solution.

To be able to compare the source code a similar application is used. The second application is a newer iteration of the mortgage broker application, henceforth called application **B**. During the implementation of this application a large part of code of application A was reused. The difference is that the new program had to be able to work with multiple products. To achieve this parts of the code were rewritten and designed to be more modular. A lot of unnecessary code was removed from the original source code.

5.2 Finding software scalability bottlenecks

The problem identification starts with identifying already available resources. The application has gone through multiple releases and has been in use for some time. All of the potential methods from the previous chapter are available for use. There are server log files, bug tracking logs available, developer experience and performance test available on request. For initial identification the log files and developer experience are used. Identified bottlenecks are then reproduced/verified with performance testing.

Log file analysis

From the server logs a top ten of bottlenecks are identified. The server logs provides an overview of the number of times a page is requested and the average time the server spent building the page. The multiplication of these variables is the total time a server spent on generating a certain page. From these logs different rankings can be created, for example:

1. A list of the pages with the highest number of requests
2. A list of the pages with the highest average response time
3. A ranking of the pages with the highest multiplication (number of requests * average creation time). This is shown in Table 5.1.

The tops of all lists are important to consider for additional research. For this research the *multiplication time(min)* row is used as source for scalability bottlenecks. The number one problem from the logs is the *Flattering*. The page with the highest average is also a point of interest. In this case this is *UploadScanDocument*. This page is not considered as a scalability problem in this situation. The limiting factor of this page is the connection speed of the user, not the code of the application. A complete overview can be found in Appendix B1.

URL	Average # request	Average response time (sec)	Multiplication time (min)
/WF/A/Flattering.aspx	14880	2,52	616
/WF/A/DocTypen.aspx	41695	0,85	560
/WF/A/PersonalData.aspx	47639	0,68	557
/WF/A/CustomerSearch.aspx	46501,5	0,71	551
/WF/TP/AanvraagSnelToets.aspx	25852	0,96	412
/WF/TP/ResultaatAlternatief.aspx	28026,5	0,86	400
/WF/A/Workspace.aspx	60640	0,38	388
/WF/TP/AanvraagOnderpandGeenDepot.aspx	13150	1,65	373
/WF/A/Offer.aspx	32018	0,69	370
/WF/A/UploadScanDocument.aspx	2702	6,32	285
/WF/A/BKRtoets.aspx	14582	1,21	284

Table 5.1: Log information from two months averaged per month

Bug tracking system

The bug tracking logs provided insight into the bug history of the application. When assessing the bug logs it is important to understand the context of the bugs. Topicus uses different categories to group bugs. These categories are linked to functions of the application, not the actual pages (like the server logs). This means the number of bugs per category is linked to functionality, not to specific pages. The miscellaneous category has the largest share but is hard to link to specific parts of an application. The other four categories are more specific. An interesting observation is that all four categories represent functions that are at the top of the server log information. The highest non-general area is once again the *flattering*.

Category	Percentage (%)
Miscellaneous	31,82
Flattering	12,30
Requested Documents	7,89
Request result	6,82
Offers	6,82

Table 5.2: Bug tracking log

Expert Knowledge

Three interviews were held with developers of the web application. Two of them with software designers that recently worked on the application and one with a software designer that worked on the original code. The interviews consisted of three main topics, possible scalability bottlenecks of the application, the modularity of the application and the reusability of the application. Possible scalability bottlenecks were mentioned to several different areas. The most mentioned part was the flattering rule engine. This rule engine is a combination of a lot of different database calls, a lot of different comparison operations and execution logic. Any of these three parts is a potential source of scalability problems. The problem is that all developers mention this may be hard in the current implementation. Most of the problems mentioned are also found in the top bottlenecks from the server logs (see Table 5.1).

Another bottlenecks mentioned was the dependency on third-party software. At some points the application needs information from third-party software. If that third party is unable to sustain the information at the rate of the Topicus application a bottleneck if formed. The problem with this bottleneck is that it cannot be solved by Topicus directly (unless they are willing and able to provide the service in-house). Other mentioned bottleneck were points in the software that either require a lot of CPU power and/or a lot of database information. Both are an important source of bottlenecks.

The modularity of the good is perceived to be quite good. Topicus designs their application to be reusable. To achieve a high reusability the code is made modular. Breaking different functions down into separate modules creates a flexible code base. The **FORCE** framework is an example of this reusable code base. For future projects Topicus is always trying to see which parts can be reused and which parts of the code can be transformed into a new module. This expansion is an important part of the development strategy.

The modularity is linked to the reusability of the code. Modular parts of the code can easily be used in new projects. In order to be able to reuse the code the developers all mentioned potential problems. To be able to effectively reuse code is has to have a clear and identifiable name and has to be documented. If a method cannot be found when searching to the code for a specific method a developer will create a new one. This means the code base grows unnecessary. And if parts of the code are reused that are not used a lot of legacy code is created in the new application. During the implementation of a new application based on the before mentioned web application a lot of legacy code was removed. This can be verified when looking at the lines of code of the two application, thousands of line of code were removed. The results was a much cleaner and easier to read code base.

In order to improve the modularity and reusability of the code all developers ment on the need for a more structured approach to code implementation. Design guidelines provide a check list that a developer can use to verify if his/her code agrees to the set standard. Naming and commenting guidelines in particular can ensure a better code quality.

Scalability testing

From the previous information sources a couple of bottlenecks were identified. The three identified scenario's were:

1. *Doing a customer search:* Searching through all the customers records for a top 15 or a specific customer by name
2. *Doing a flattering:* The execution of the rule engine and logic using all kind of information from the mortgage to calculate which mortgages are available to the customer
3. *Loading the Workspace:* After a login a overview of available propositions is generated

The three different scenarios were combined into a web test. A web test is a recording of a user executing different actions. During the web test the followings steps are taken:

1. The user logs in
2. Searches for a specific customer using the customer search and open a proposition
3. Does a flattering on the proposition
4. The users logs out

The scenario was recorded with Visual Studio 2008. From the web test a load test was generated. The load test is a execution of the web test using 1, 25, 50 and 100 concurrent users during two iterations. One lasting 10 minutes and one lasting 15 minutes. Fifty percent of the users had a simulated T3 connection and other fifty percent used a T1 connection. This was done in order to avoid pushing the LAN connection speed to its limit and make a more realistic simulation. The load test was done on a local acceptation server of that specific application.

In all the load tests, the flattering was the greatest bottleneck when increasing the number of concurrent users. This is shown in Figure 5.1. The average page time increase of the flattering page is large and the CPU time on the servers is extremely high. There are no page results from the 100 concurrent users test because within the time limit only "request timed out" messages where returned from the server. The full results can be found in Appendix B2.

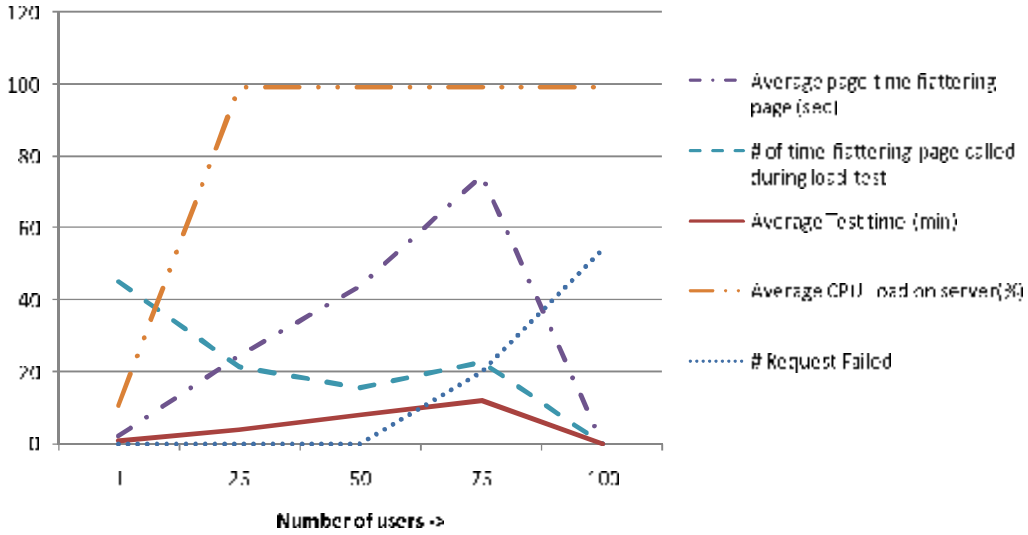


Figure 5.1: Result of the load tests

Bottleneck ranking

The most mentioned scalability issue identified by the different methods and tools is the *flattering*. The flattering is an important part of the application and used throughout this and other applications. It is also already part of the reusable FORCE framework. This makes this part of the application an interesting subject for further analysis.

5.3 Analysis software scalability bottlenecks

The different bottleneck identification steps lead to the identification of several different bottlenecks. The flattering was identified as the current weakest part of the application. In this section this part of the application will be further evaluated. It is difficult to look at environmental flexibility from only a small part of the code. In this case the entire application is evaluated. In the next parts different proposed methods and tools results are discussed.

Code Analysis

The flattering of both applications were analyzed with a profiler. The profiler creates a call tree of the execution of the flattering of application A and B. Both call trees consist of two bigger methods and some smaller ones. The two big methods are GenerateAllDocuments and Flattering/Flatteer. In application A GenerateAllDocuments took around 76% of the thread time. In contrast Flattering uses only 17% of the time. Application B is more balanced. Flatteer takes around 48% of the time and GenerateAllDocuments around 52%. A more detailed overview of the profiler is given in Appendix B3.

The profile of application A points out more interesting points. A number of methods is called more than one time, when only one call is expected (in the case of data retrieval). The method seems to be called four times without an obvious reason. And multiple unnecessary calls take up time and space that decrease the scalability of the application. Multiple executions can also have other unintended effects. In case of application A it increases the chances of an autoflush. During the interviews one of the developers mentioned the autoflush which is reload of the information from the database. The less calls being made the lower the change of an autoflush. The flattering of application B has less multiple calls which improves its scalability. From this test the flattering of application B can be considered more scalable.

Code Metrics

The environmental flexibility of the code was expressed in terms of different software metrics. The first four metrics coupling, cohesion, complexity and instability are related to the modularity of the code. The investigated code consisted of all the code that was specifically written for the application. Any supporting framework code was not evaluated in this step. The full metric results can be found in Appendix B4.

Application B has lower amount of afferent coupling (incoming dependency) and a higher efferent coupling (outgoing dependency) than application. The average relational cohesion of application A is lower than that of application B. As mentioned in the previous chapter the optimal range for cohesion is between 1,5 and 4. Both are within this range so can be considered cohesive enough. But on average the modules of application B are more cohesive than those of application A. The instability of application B is slightly lower which means they are on average a bit more stable than modules of application A. Instability occurs when there is a difference between the afferent and efferent coupling of an application. Instability is an indicator of resilience to change. The Cyclomatic Complexity of application B is a lot lower than that of application A. During the redesign of application B a lot off unused code was removed. This also removed a lot of different paths through the application which translates into a lower complexity.

	Total Afferent Coupling	Total Efferent Coupling	Average Relational Cohesion	Average Instability	Cyclomatic Complexity per class
Modules of application A	725	756	2,26	0,71	12,46
Modules of application B	706	985	2,80	0,67	11,33
Difference	-19	229	0,54	-0,04	-1,13

Table 5.3: Code metrics overview of modules of application A and B

Maybe more interesting is the difference in between the modules used by the flattering of both applications. The profiler identified the modules used by both flatterings. The modules of application A that use the flattering have a higher number of afferent coupling and a lower number of efferent coupling than those of application B. The relational cohesion application B is also higher, but is still within the optimal range of 1,5 and 4. This means that the modules that are used by the flattering of application B are more cohesive than those of application A. Application B has a slightly higher instability. This means the modules used by flattering of application A are slightly more resilient to change than those of application B. The complexity per class of application B is lower which means the code is less complex.

	Total Afferent Coupling	Total Efferent Coupling	Average Relational Cohesion	Average Instability	Cyclomatic Complexity per class
Modules of application A used by flattering	715	592	2,60	0,51	15,25
Modules of application B used by flattering	642	648	2,96	0,59	13,94
Difference	-73	56	0,36	0,08	**-1,31**

Table 5.4: Code metrics overview of modules of application A and B used by flattering

Dependency Structure Matrix and Graph

The Dependency Structure Matrix (DSM) of the application gives information about the dependencies between different parts of the application. A complete overview of the dependencies on assembly level is given in the DSM appendix B. When investigating the dependencies of a web application the dependent modules can be divided into two categories:

1. *Application modules:* Modules that are "unique" to the application and encompass the application code
2. Framework modules: Library modules that are reusable among multiple application without change. Within this category there are generic .NET framework modules and the reusable Topicus modules as the FORCE components and Topicus Library

The FORCE and Topicus modules are the sources of generic and reusable code. The FMS modules FMS, FMS.Jobs and FMS.Templates only have outgoing dependencies (they are on the left side of the DSG). This makes modules unstable because they are dependent on a number of other modules. A change in one of these assembly can have an unintended effect on the higher level modules. The modules with mostly incoming dependencies are seen as stable. The most stable modules of the web application are the more generic reused Topicus framework modules as their FORCE components and their library. This is also shown in the dependency graph of appendix B5.

Software reusability

The average documentation of application A is almost 16% (see Table 5.5). This amount of code documentation is considered low by industry standards. As mentioned in chapter 4 optimal percentage of documentation is between 20% and 40%. This is consistent with remarks from the interviews. Different developers already commented on the necessity for more and qualitatively better documentation of the code. This metric however does not give any insight in the quality of the documentation. The documentation does not only need to be available but also of good quality. There is however no good automated tool or method to measure these metrics.

Total	LOC	Comments	Comments (%)
Application A	199.628	37.580	15,84%
Application B	148.070	28.360	16,07%
Difference application A -> B	-51.558	-9.220	0,23%

Table 5.5: Global source code overview

The reuse metrics give information about the current reuse and reusability of the code between application A and B. Almost 39% of the code has been maintained between the two version. The other 61% percent of the code has been changed. Almost 50% of the code is completely new and the other 11% are inline changes (see Table 5.6). The complete overview can be found in Appendix B4.

Application A -> Application B	LOC	Comment
New lines of code in application B	73.260	16.063
Removed lines of code from application A	-124.818	-25.283
Equal lines in both projects	57.002	10.113
Inline changes	17.808	2.184
Total for Application B	148.070	28.360
Direct reuse in Application B (% equal)	38,50%	35,66%
Percentage changed	61,50%	64,34%

Table 5.6: Detailed reuse metric overview

Bottleneck analysis

The flattering is a complex part of the application that is coupled to a lot of other modules of the application. A comparison of the application code of the two application shows that application B can be considered more modular because of its higher cohesion. The reuse metrics are also more favourable for application B. This is not a surprise because the old code of application A was rewritten to be more flexible. The removal of unnecessary code in application B increases the ability to read and reuse the code in different situations. This also lowers the average complexity per class.

Another side effect of the changed code is the increase in dependencies within the application in application B. The flattering needed to be more flexible new links were needed to be able to deliver that flexibility. In this case the increased dependencies are not necessarily a bad thing because this particular function is more easier the reuse when it is flexible and not static as it was in the old situation.

5.4 Software scalability bottleneck solution strategy

The flattering is an important but difficult part of the application. It executes a large number of business rules against a lot of information from the database and is used multiple times throughout the important use cases of the application. The software scalability can be improved on two points:

1. The database connection
2. The execution of the code

The first solution is not considered because this research does not focus on the database. The document generation part of the flattering is the more obvious choice for revision because it takes up the largest part of the flattering. As previously mentioned the multiple execution of data retrieval is a performance bottleneck. Different data is reused throughout the flattering but retrieved multiple times. Caching of this data removes the database calls and improves the load of the application and the database.

A more radical solution would be the removal of the flattering from the application. If the flattering code is highly modular with high cohesion and low coupling it could be transformed into a separate web service. This separate web service has the advantage that the specific needs of the flattering can be met without comprises to other parts of the code. The flattering has a high CPU demand and a high database load. Separating it means the service can potentially be used by one or multiple servers and can be scaled for greater demand.

This separation is not without the creation of additional overhead. Exchanging data between machines requires more control than sharing data locally. If the web application can be shared by multiple servers the additional overhead is expected to be lower than the overhead required for using multiple servers each running the entire application.

During the interviews the experts mentioned that optimizing the code of the flattering of application A might be difficult in its current implementation. The current implementation cannot be considered flexible(this is also shown from the code analysis). For application B the flattering was implemented to be more flexible. The difficulty of a flattering implementation is that the exact implemencation may differ per application. While general idea of the flattering is the same among multiple projects, the specific business rules are different. A truly scalable solution might be a more abstract meta-engine with meta-rules that can be easily adapted to the specific applications.

6. Discussion

This chapter is a discussion about the contributions and results of this thesis so far. The first part of this chapter is a discussion of the idea and usability of software scalability. What are the advantages and differences of software scalability, and how can it be used during development? The second part is a discussion of the evaluation model. During the case study different methods and tools were used. Some of these methods worked and some did not. The discussion will talk about the findings in greater detail.

6.1 Software scalability

Software scalability is a versatile concept and this thesis further refines the concept. Scalability can be improved along two dimensions. The first dimensions is the vertical scalability of an application. Creating vertical software scalability is the ability to optimize a single application to be able to serve more users and/or work. This is defined as the *scalability optimization* of an application. The second dimensions is the horizontal software scalability of an application. Horizontal scalability is the ability to create an application that is usable in different environments and multiple systems. This is defined as the *environmental flexibility*.

Traditionally researchers only focus on one of these dimensions of scalability. This means most of the research only talks about the optimization of a single application or the distribution among systems. One of the key points of this thesis is focusing on scalability along both dimensions. Thinking about scalability of software means thinking about both dimensions. Some application will have a greater focus on the optimization of the application, where others applications focus on being flexible. This does not mean one should only focus on one dimension and forget the other. Both are important in creating scalable software. Optimizing a single application works as long the expected number of users can be supported by a single application. When the workload grows larger than the application can handle, a second application is needed. If the application is not flexible enough the cost for porting the application to a distributed setup will be very high. If an application is focused on both dimensions it will be both usable as a single application but also usable in a distributed setup.

To create software scalability changes have to be made to the application code. Application code is changeable if a company has both the resources (man hours and expertise) and the intellectual ownership (IP) of the code. Having both enables someone to make changes to code. Which changes to make is dependent on which dimensions of scalability one wants to improve.

The return of investment from software scalability comes from creating scalable (parts of) software. This reuse can be in the form of both application code and/or knowledge. This in turn means investments in software scalability are usable among different iterations and instances, which means the gain in quality and decrease in investment increases the ability for return on investment. As previously said generating knowledge and awareness of quality is also an important part of exploring software scalability. Sometimes specific code is not directly usable, but the idea behind the code is. An example of this could be incompatibility between Java and C# code. They cannot be interchanged directly but the function of the code can be rewritten in another language.

It is also important to not see scalability evaluation as a onetime investment. Scalability is a weakest link problem that is not completely solved in one iteration. Scalability bottlenecks can be found in both software and hardware and both require different solutions. A long term continual investment adds to a larger knowledge and code base about scalability. Especially if the investments and results are shared between multiple organizations a good investment/return ratio can be achieved.

In the end scalability is both a hardware and software issue. True scalability can be created by creating a synergy between both sides. An organization that wants to improve scalability should consider both options. Both are usable in different situations and both have their own advantages and disadvantages. If the two are used in balance they can improve scalability beyond what only one of them could.

6.2 Software scalability evaluation model

The goal of the evaluation method is to find and solve scalability bottlenecks in software. This is different than most of the existing methods for scalability. Most of the existing methods are a comparison for two situations. The proposed method is about finding and solving problems. The main goal of the evaluation model is guiding the user in obtaining information about the scalability of the software. This information is needed to see which part of the code can be improved. By using the different tools and methods more insight in the application can be generated which can help to select new and different tools and methods to gather even more precise data.

The three steps of the evaluation method are generic and widely applicable in different forms. During the case study the evaluation method was used to identify scalability bottlenecks in a current application of Topicus and use an analysis to give recommendations for a solution. The first step is the identification of bottlenecks in the application. Using different tools and techniques a number of bottlenecks were identified. What worked was the ability of the different methods to find scalability bottlenecks using different sources. Both the log files and the expert interviews identified similar problems. And load testing was a method to validate these problems by active testing. The limitations of these steps are that they are primarily usable to identify scalability bottlenecks in application (scalability optimization).

The second is the analysis of the bottlenecks identified in the first step. In its current implementation the analysis is somewhat limited. The tools, methods and metrics that were chosen represent a small portion of what is possible. Both dimensions of software scalability have their own characteristics. Scalability optimization is mostly measured using only the application under inspection. The profiler tool, the dependency matrix and the cyclomatic complexity and cohesion metrics were chosen to represent the internal structure of the code. Environmental flexibility is not easily measured on its own. It needs a similar application to provide the necessary insight into which characteristics works well and which does not. The modularity and reuse metrics were chosen to represent this ability.

The goal was to, similar to the first step, be able to combine the different sources into a fitting overview of the application. The main problem is that the different method and tools were not meant for evaluating software scalability as a whole. Some tools and methods are usable for the scalability evaluation and other for the flexibility and modularity of the code. Not all the results of the methods and tools could be linked to scalability or to another metric that does link to scalability.

Especially the software metrics were not as usable as hoped. The biggest problem is that not all metrics could be linked to scalability. The metrics were chosen to represent certain elements of both horizontal and vertical scaling. Application B is newer and more flexible and modular version of application A. In this sense it was expected to have better results for scalability than A. However, this could not been distilled from the results. While the better results for the reuse metrics and less complexity of application B indicate that it is indeed more modular and flexible the evidence was not strong enough to indicate a real impact on scalability. For future evaluation perhaps other metrics are more suitable for evaluating horizontal scalability. This is further discussed in the recommendations section of chapter 7.

The comparison of the flattering was hard to evaluate because of the differences in application things are not always directly comparable. The changes in coupling are also difficult to evaluate. Application B was build to be flexible, one way to accomplish this was by a greater separation of concern. This increased the number of modules of application B. And the greater number of modules increased the total number of metrics (see Table 5.3). The idea was that more coupling meant lesser scalability. In this case application B is considered more scalable but the greater amount of coupling does not seem to be that bad or have a big impact on scalability. The instability metric is dependent on the amount of coupling. If coupling cannot be linked to scalability the instability metric is also not usable in this situation.

The third step is the solution strategy for the bottleneck. Dependant on which dimension is of more interest to the solution, the specific solution may change. This thesis is not interested in the specifics of the solutions these are not evaluated. Just as important as the actual solution is the reuse of the solution. Optimized code can, if flexible enough, be reused among similar implementations. Especially is the reusable code is part of generic components, for example from a library component the code is easily distributed among projects. In case the code is not directly reusable, knowledge from the solution can be reused. In case one program is build in C# and another in Java the code cannot be reused directly. Even if the code is not directly reusable the design and logic of the solution is, all it needs is somebody who can implement the solution in Java.

The case study at Topicus showed that valuable information can be gained from using a structured approach for the identification and improvement of scalability. An overview of which methods and tools worked and which did not are given in Table 6.1. The biggest problem areas are the metrics that measure horizontal flexibility. While the metrics are usable for measuring portability and code flexibility, this does not mean it measures horizontal scalability.

Test tool/method		Usable for evaluating scalability?	Measures vertical/horizontal scaling ability?
Log files		Yes	Vertical
Bug tracking system		Yes	Vertical/Horizontal
Expert knowledge		Yes	Vertical/Horizontal
Load testing		Yes	Vertical/Horizontal
Profiler		Yes	Vertical
Dependency Structure Matrix and Graph		Yes	Vertical
Software Metrics	*Coupling*	No	Horizontal
	Cohesion	Undetermined	Horizontal
	Complexity	Undetermined	Horizontal
	Instability	No	Horizontal
	Reuse of code	Yes	Vertical/Horizontal
	Documentation rate	Yes	Vertical/Horizontal

Table 6.1: Software evaluation method overview

Further improvements can be achieved by using better suited methods and tools. This specific case study focused on the scalability of web applications. While scalability can be considered more important for web applications than traditional application, traditional applications can also benefit from being more scalable. Different tools and methods may be appropriate for the evaluation of more traditional applications. Tools and methods can be added or removed because of two reasons:

1. The goal of the method is gathering information about software scalability. This information is needed for deciding further actions. Without this information no further actions can be taken. How this information is gathered is not important for the future actions. The precise method and tool used are of lesser importance. This research has not investigated every possible method and tool. The goal was to give some options for increasing software scalability but is by no means complete overview.
2. At the moment there are no tools and methods that are completely compatible with the newly proposed definition of software scalability of this thesis. This means tools and methods can be improved for better information gathering.

The evaluation method can also be used for evaluating other characteristics. The three step model is generic enough for answering different questions. In that case other methods and tools can be used to gather the requested information. Some existing methods as the expert knowledge can be used in every situation.

7. Conclusion and Recommendations

This final chapter discusses the conclusion and the future of this research. The first part is a summary of the entire research and its most important and interesting contributions to the knowledge about scalability. The second part consists of recommendations for both Topicus and future research. This research is only an exploration and there are still many interesting directions.

7.1 Conclusion

In this thesis scalability is defined as the *ease of expending an application to serve more users/and or work*. For web applications scalability is an important factor. A web application is a centrally hosted application on which people from various locations login to use the application. A web application has to handle a dynamic amount of users its ability to handle a variable workload is important.

The first part of this thesis was about the notion of software scalability and how it relates to the larger issue of general scalability. The biggest problem of scalability currently has is that it is a large and complex subject without a good scientific basis. It has no clear and accepted definition and this makes the research fractured. Different papers talk about different aspects of scalability and mostly define scalability as a hardware issue, or a system issue. This makes the research fractured and difficult to use.

What makes scalability so complex is that it is a something that is influenced by a lot of other characteristics/circumstances. This is further complicated by the behaviour of scalability. Scalability is a problem that cannot be "*solved*". Like other quality characteristics it is improved until it becomes acceptable or meets stated criteria but in theory it could potentially be improved further. On the practical side there is often a trade-off between the advantages of the improvement and its costs. Scalability improvement can be seen as a weakest-link improvement. If you want to improve scalability you start with the weakest part of the chain. After that part is improved another part becomes the weakest link.

The focus on software scalability makes this thesis stand out against the current literature. Instead of only thinking in adding servers or replacing hardware, scalability can also be created in software. This thesis further divides software scalability into two dimensions:

1. *Software scalability is the ability to handle increased workload by changing parts of the code (scalability optimization/vertical software scalability)*
2. *Software scalability has the ability to be used multiple times in a cost-effective way (environmental flexibility/horizontal software scalability)*

These two dimensions explain the two most important characteristics of achieving software scalability. The main advantage of analyzing and improving scalability on the software side, instead of the hardware side, is that the solution is reusable. This changes the way the scalability of software should be analyzed. This means scalable software should not only be able to handle an increased workload (*scalability optimization*) but also be reusable in different environments (*environmental flexibility*). Combining these aspects results in scalability optimizations that are usable in multiple situations and iterations.

This does not mean an organization should only focus on software. A truly scalable system is dependent on both hardware and software. This author does not believe one is more important than the other. Both have their own strengths and weaknesses and both are usable in different situations. An organization that is wants to invest in a scalable system will need to create a synergy of the best elements on both sides.

An iterative approach that consider both hardware and software also fits with the weakest link behaviour of scalability. If an organization has a wide range of possibilities it can fix the weakest link with the most appropriate solution. An example of this is improving the system iteratively along the different dimensions. A first iteration would be improving vertical software scalability, the second iteration would be improving horizontal software scalability and the third would be adding and/or replacing hardware, and so on.

The second part of the this is an evaluation method to help an organization in identifying and analyzing bottlenecks on the software side. To increase software scalability an evaluation method is proposed which can be filled with various tools and method. The evaluation method proposes a three step model:

1. The identification of software scalability bottlenecks
2. An analysis of software scalability bottlenecks
3. A solution strategy for software scalability bottlenecks

The first two steps consisted of a number of different tools that were chosen to measure different aspects of the horizontal and vertical scaling ability of software. The three step model was evaluated by applying it to a Topicus web application. For some methods and tools a second application was introduced as a comparison for the first application.

The first step consisted of using information from log files, bugtracking systems, expert knowledge and load testing to indentify scalability bottlenecks. The four methods/tools identified several, both similar and different, bottlenecks. The different methods/tools complemented each other and the load testing offered a way to verify problems identified by the other methods/tools.

The second step consisted of code analysis and code metrics. A profiler was used to analyze the execution of the code and a dependency analysis was used to analyze the dependencies between different parts of the code. The code metrics coupling, cohesions, complexity and instability are part of the code structure analysis, the reuse of code and documentation tells are indicators of the reusability of the code.

The profiler, dependency structure matrix and reuse metrics all showed information that could be usable for better understanding the scalability of an application. These metrics/tools all showed, mostly, information about the vertical scalability of an application. Evaluation of horizontal scaling proved to be more difficult. The case study results of coupling and instability did not match the expectations. In this case they could not be linked to scalability. The other metrics as complexity and cohesion showed promise in indicating scalability. This however could not be proved empirically during this research.

The final step is thinking about the solution to the bottleneck and how the improved code can either be reused directly or reused as knowledge. The reuse is important because otherwise it will be hard to justify the investment in code changes.

Almost similar steps were also used by Topicus in another situation. During this research one of the applications had a performance issue. For a solution parts of the code and logs were used to identify the most pressing problem. Knowing the biggest point of concern should lead to further analysis. In this particular case the database was the point of failure. During runtime the load on the database was uncommonly high. Further investigation showed that the database had a hard time executing all the different queries on the records. The solution in this case came from revaluating the indexes placed on the database. This resulted in a much lower database load.

This situation allowed Topicus to gain knowledge about this kind of bottleneck. Topicus is now able to identify and optimize this kind of bottleneck in the future. The bottleneck solution was also shared internally and can now be used by all Topicus organizations. A similar scenario is recommended for scalability issues.

These steps can be divided into the three steps of the evaluation method. In this case the problem was performance related and because of this the implemented tools and methods are different but the basic steps remain the same. This makes integration of tools and methods that indicate scalability easy in an organization as Topicus. Other organizations might have other needs of possibilities. Organizations are encouraged to tailor the evaluation method to their own specific desires.

Not all methods seemed to indicate scalability clearly. For future use new and different methods can be used during the first two steps. In case an organization wants different information different methods can be used. Especially metrics that are more tailored for software scalability and the evaluation of horizontal and vertical scaling of software will increase the quality of the information gathered during the evaluation steps. Future steps are further considered in the next section.

7.2 Recommendations

The recommendations are separated for Topicus and for future research. Topicus is interested in a practical approach for evaluating and improving scalability. On the research side there is still a lot of additional research options. This thesis is an introduction into the field of software scalability. More research is needed for further investigation and validation of this field.

7.2.1 Recommendations: Topicus

Topicus as a software developer has a good position for improving scalability on the software side. As a software developer they have the experts and knowledge in-house to find and solve bottlenecks in software. As creators, and owners in case of a SAAS application, they have access to the source code. If the application is web based the software can be easily updated because there is no need to redistribute or patch the application on the client side.

The Topicus development method also fits the software scalability approach. This method is already oriented on software reuse and modularity. This proposed approach to software scalability also recognises these points and marks them as important. If scalability awareness and optimization can become part of the development cycle code and knowledge can be effectively reused in other projects and iterations.

This leads to the primary recommendation for Topicus:

Create awareness of (software) scalability and promote the reuse of code and knowledge throughout the organization

The importance of quality is growing within Topicus and scalability should be one of the important quality factors. The ability to handle scalability effectively and cost-effective can be a competitive advantage in the web application market.

Knowledge sharing can be promoted by creating places and events for information exchange. This can be done by writing best practices and guidelines about scalability, hosting discussion events and freeing up resources for additional scalability research. This research is done at a high level and does not give any concrete solutions. An opportunity for further research, without taking too much time away from employees, would be internships that investigate more specific scalability solutions. For example research into database optimization, load balancing and method complexity.

Scalability is not only an issue for Topicus Finance but is of importance to all Topicus organizations. This means sharing knowledge between organizations creates additional opportunities. As mentioned before, scalability can be improved on many different levels and places. Creating a large knowledge base of solutions for different languages, architectures and environments can improve applications across all organizations. Within a single organization reusing code that is scalable can speed up development time and assure a certain level of (software) scalability. Being able to share information between the different Topicus organizations provides even better opportunities knowledge generation and knowledge sharing.

7.2.2 Recommendations: Future research

Scalability is an area that has a lot of research potential left. Especially in the area of software scalability there is a lot additional research to be done. This thesis is an exploration into the definition and use of software scalability and only scratches the surface of the subject. Future research is needed both in detail and across multiple disciplines.

The proposed definition of software scalability is still somewhat broad. The definition is a statement of the goal but leaves its implementation to the user. While this research has tried to define every aspect of software scalability some points may have been missed or require more detailed investigation.

The overall structure of the evaluation framework is adaptable for different and more analyses methods and tools. The current implementation is based on a small collection of available methods and tools. As mentioned in the previous chapter not all results are easily combined into a clear end result. For a big part this is because there is no real previous research into software scalability as a separate area. This means there are no methods and tools proposed or validated that completely fit the definition of software scalability. Maybe current tools and method can be adapted for use but this is beyond the scope of this thesis. A better understanding of this area of scalability and a better support by tools and methods will make it easier to identify, analyze and solve scalability issues.

One of the more interesting points is the evaluation of software scalability using metrics. For the evaluation of the environmental flexibility a number of metrics are proposed. More research is needed to validate this theory behind the link between these code metrics and the presence of software scalability. Research into other, more direct methods and tools for software scalability evaluation is also encouraged.

Especially measuring horizontal scalability proved to be difficult, new insights into this characteristic can greatly improve the measurement and improvement of software scalability. The definition of horizontal scaling is similar to that of *portability*, *code understandability* and *reusability*. The goal of these methods is the same as horizontal scaling, creating flexible software that can be adjusted to a new or changed environment. New metrics and tools for horizontal scalability can be searched in that research area. A number of articles about the reusability define metrics for evaluating the reusability of software. Portability and understandable code are often a part of this research area. More extensive reusability metrics will hopefully provide a better insight into horizontal scaling [40],[57].

The results of the empirical test on vertical and horizontal scalability test have shown it is still hard to quantify scalability as a whole. The current evaluation method approaches scalability from a number of different angles for a broader overview. And as mentioned some angles are better quantifiable as others. All the metrics showed quantifiable information about the application code, the problem was linking that information to scalability. Some metrics, as cyclomatic complexity and cohesion, showed promise in measuring software scalability. This however could not be confirmed during this research. Future research might be able to identify a better and stronger link between complexity, cohesion and scalability.

As mentioned before, software scalability is only a part of scalability in general. To further improve software scalability the greater idea of scalability also needs more research. One of the biggest problems is for future research is the absence of a well accepted definition for scalability. Without an accepted definition future research is likely to focus on small incidental research areas without an idea of how this relates to this big picture, or what is missing in that picture. Another risk is that a change in of definition may remove or change the applicability and validity of certain scalability research.

Future research should not only further explore the definition and application of software scalability but also its link to the greater scalability issue. Scalability is both software and hardware issue. An application is dependent on both, and both are usable for different situations. An interesting point of research is which situation requires which approach. The investment needed for the scalability improvement is also a factor in the decision. A direct comparison and calculation method for hardware versus software may lead to a decision support method.

References

[1] A. Abran, W. M. James, P. Bourque, and R. Dupuis. *Guide to the Software Engineering Body of Knowledge 2004 version*. IEEE Press., 2004.

[2] K.S. Ahluwalia. Scalability design patterns. In *14th Conference On Pattern Languages Of Programs*, 2007.

[3] R. Ahmad, Zhang Li, and F. Azam. Web engineering: a new emerging discipline. In *Emerging Technologies, 2005. Proceedings of the IEEE Symposium on*, pages 445–450, 17-18 Sept. 2005.

[4] F. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the iso/iec 9126 quality standard. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, page 7pp., 17-18 Nov. 2005.

[5] Vicenc Beltran, David Carrera, Jordi Torres, and Eduard Ayguade. Evaluating the scalability of java event-driven web servers. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 134–142, Washington, DC, USA, 2004. IEEE Computer Society.

[6] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[7] B.W. Boehm. *Characteristics of Software Quality*. North Holland, Amsterdam, 1978.

[8] A.B. Bondi. Characteristics of scalability and their impact on performance. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, New York, NY, USA, 2000. ACM.

[9] Encyclopedia Brittanica, 2007.

[10] Dov Bulka. *Java Performance and Scalability: 1 (The Sun Microsystems Press Java Series)*. Addison Wesley, 2000.

[11] E. Burris. Programming large website by eddy burris, 2007.

[12] L. Chung, B.A. Nixon, Y. Eric, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering (THE KLUWER INTERNATIONAL SERIES IN SOFTWARE ENGINEERING Volume 5) (International Series in Software Engineering)*. Springer, October 1999.

[13] Y. Deshpande, S. Murugesan, A. Ginige, S. Hansen, D. Schwabe, M. Gaedke, and B. White. Web engineering. *Journal of Web Engineering*, 1(1):3, 2002.

[14] T.J. Douglas and W.Q. Judge jr. Total quality management implementation and competitive advantage: The role of structural control and exploration. *Academy of Management Journal*, 44(1):158–169, 2001.

[15] L. Duboc, D. Rosenblum, and T. Wicks. A framework for characterization and analysis of software system scalability. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 375–384, New York, NY, USA, 2007. ACM.

[16] Topicus Finance. Force framework whitepaper, 2006.

[17] W.B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Softw.*, 11(5):14–19, 1994.

[18] W.B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, 2005.

[19] H. Freeman. Software testing. *Instrumentation & Measurement Magazine, IEEE*, 5(3):48–50, September 2002.

[20] D. Garvin. What does "product quality" really mean? *Sloan Management Review*, 26(1):25–43, 1984.

[21] P. Gerrard. Risk-based e-business testing: Part 1 – risks and test strategy. Published on WWW by Systeme Evolutif Ltd., 2001.

[22] A. Ginige and S. Murugesan. The essence of web engineering - managing the diversity and complexity of web application development. *Multimedia, IEEE*, 8(2):22–25, April-June 2001.

[23] Google. Google seattle conference on scalability http://www.google.com/events/scalability_seattle/, 2007.

[24] D.B. Gustavson. The many dimensions of scalability. In *COMPCON*, pages 60–63, 1991.

[25] H.D. Harlow and S. Imam. The effect of tacit knowledge management on innovation: Matching technology to strategies. In *Technology Management for the Global Future, 2006. PICMET 2006*, 2006.

[26] M.D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.

[27] I. Im and A. Hars. Knowledge reuse — insights from software reuse. In *Proceedings of the 1998 Americas Conference of the Association of Information Systems (AIS)*, 1998.

[28] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(6):589–603, 2000.

[29] B. Kitchenham and S.L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996.

[30] C.W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[31] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *J. Parallel Distrib. Comput.*, 22(3):379–391, 1994.

[32] S.i Lai and C. Yang. A software metric combination model for software reuse. In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 70, Washington, DC, USA, 1998. IEEE Computer Society.

[33] E.A. Luke. Defining and measuring scalability. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 183–186, 1992.

[34] J Markoff and J. Hansell. Hiding in plain sight, google seeks more power. New York Times, June 2006.

[35] T. McCabe. A software complexity measure. *IEEE Trans. Software Eng.*, 2(1):308–320, December 1976.

[36] J.A. McCall and M.T. Matsumoto. Software quality measurement manual, vol. ii. Technical report, Rome Air Development Center, 1980.

[37] J.D. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. Performance testing guidance for web applications: patterns and practices. Microsoft Corporation, 2007.

[38] J.D. Meier, S. Vasireddy, A. Babbar, R. Mariani, and A. Mackman. *Improving .Net Application Performance and Scalability*. Microsoft Corporation, 2004.

[39] A. Mishra. Performance scalability in switching system software. *Communications, 1999. ICC '99. 1999 IEEE International Conference on*, 1:290–295 vol.1, 1999.

[40] J.D. Mooney. Portability and reusability: common issues and differences. In *CSC '95: Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pages 150–156, New York, NY, USA, 1995. ACM.

[41] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[42] NDepend. Ndepend metric definitions - http://www.ndepend.com/metrics.aspx/, 2007.

[43] F. Nickols. *The knowledge management yearbook 2000-2001 - The knowledge in knowledge management*, chapter Part One - The Nature of Knowledge And Its Management, pages 12–21. Butterworth Heinemann, Boston, 2000.

[44] J. Offutt. Quality attributes of web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, 2002.

[45] J. Offutt. Web software applications quality attributes. In *Proceedings of the Conquest 2002 conference*, 2002.

[46] D. Reifer. Ten deadly risks in internet and intranet software development. *Software, IEEE*, 19(2):12–14, March-April 2002.

[47] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. National Institute of Standards and Technology, Research Triangle Park, NC, May 2002.

[48] Andreas Rudolf and Rainer Pirker. E-business testing: User perceptions and performance issues. In *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, page 315, Washington, DC, USA, 2000. IEEE Computer Society.

[49] R. Seidler de Alwis, E. Hartmann, and H.G. Gemünden. The role of tacit knowledge in innovation management. *19th IMP Conference, Lugano, Switzerland*, 1:23, 2003.

[50] S.G. Shiva and L.A Shala. Software reuse: Research and practice. In *ITNG '07: Proceedings of the International Conference on Information Technology*, pages 603–609, Washington, DC, USA, 2007. IEEE Computer Society.

[51] W. Stevens, G. Myers, and L. Constantine. *Structured design*, pages 205–232. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[52] B. M. Subraya and S. V. Subrahmanya. Object driven performance testing in web applications. In *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, page 17, Washington, DC, USA, 2000. IEEE Computer Society.

[53] B.M. Subraya. *Integrated Approach to Web Performance Testing: A Practitioner's Guide*. Idea Group Inc., 2006.

[54] Topicus. Topicus homepage - http://www.topicus.nl/, 2007.

[55] P. Verschuren and H. Doorewaard. *Het ontwerpen van een onderzoek*. Lemma, 2000.

[56] D. Wang. Meeting green computing challenges. In *High Density packaging and Microsystem Integration, 2007. HDP '07. International Symposium on*, 2007.

[57] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 211, Washington, DC, USA, 2003. IEEE Computer Society.

[58] M.B. Weinstock and J.B. Goodenough. On system scalability. Technical report, Carnegie Mellon University, 2006.

[59] E.J. Weyuker and A. Avritzer. A metric to predict software scalability. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 152, Washington, DC, USA, 2002. IEEE Computer Society.

[60] L.G. Williams and C.U. Smith. Web application scalability: A model-based approach. Technical report, Software Engineering Research and Performance Engineering Services, 2004.

[61] L.G. Williams and C.U. Smith. Qsem: Quantitative scalability evaluation method. In *Int. CMG Conference*, pages 341–352, 2005.

[62] J. te Winkel. Growing large while staying small: Spinning-off as an organizational strategy. Master's thesis, University of Twente, 2007.

Glossary

The glossary contains a shortlist of abbreviations, Topicus specific concepts and important concepts of the thesis.

Agile development methodology	A collection of software development processes that put emphasis on iterative software development. This in contrast to plan-driven development
ASP (Application Service Provider)	A provider that is capable of providing an application as a service (see SAAS)
Call tree	A tree shaped overview of the different function calls of (parts of) an application
Coupling	The amount of dependency between different software modules
Cohesion	Internal amount of dependency in a single software module
Dependency Structure Matrix	A matrix overview of the dependencies of an application
Dependency Structure Graph	A graph overview of the dependencies of an application
Explicit knowledge	A collection of recorded knowledge that is easily transferable to other people
e-Business	Conduct of business on or with help of the Internet
FORCE framework	A collection of reusable components used by Topicus to develop their web applications
Hardware scalability	Achieving scalability on the hardware side
Implicit knowledge	A collection of knowledge that could be recorded if needed
Intellectual property (IP)	The legal rights for creations of the mind as text, music, software code and such.
Non-functional requirement (NFR)	A mandatory part of an application that is not related to functional behaviour
Quality requirement	*see Non-functional requirement*
Scalability	The ease with which a system can be expanded to serve more users and/or work

SAAS (Software as a Service)	A cost model based on usage of the software not on its ownership
Software Metrics	Quantitative measurement of certain software characteristics
Software scalability	Achieving scalability on the software side
Tacit knowledge	Intuitive knowledge a person is not actively aware off
Throughput	Number of transaction per second an application can handle
Workload	Total amount of transactions an application can handle during a specific time period

Appendices

Appendix A: Preliminary investigation into Non-functional requirement

What are Non-functional requirements?

Non-functional requirement are an important influence on the software development cycle. Whereas functional requirements specify what an application should do, non-functional requirements specify how an application should perform and give specific project and development constraints. Non-functional requirement are also known in the software engineering literature by different terms. Terms as *quality attributes*, *extra-functional requirements*, *non-behavioural requirements*, *constraints* and *goals* are frequently used throughout the literature for the same notion. This thesis uses the term non-functional requirements because it is the common term in industry.

To help explain the term non-functional requirements in the software domain better a clear and formal definition is needed. The Software Engineering Body of Knowledge (SWEBOK) defines a software requirement as: *"a property that must be exhibited to solve problems in the real world"* [1]. Translating this to non-functional requirements would make it a non-functional property that an application should have.

Specific non-functional requirements are defined as aspects of the software. Most IT professionals know these aspects as the –ilities of software. The –ilities are a list of aspects like availability, scalability and the like, most of which end in *–ility*. While there is no complete list of –ilities, many of them can be found throughout non-functional requirements standards, for example ISO 9126 [7]. The extensiveness of the list, it contains more than fifty -ilities, makes it unusable as a real guideline in practice.

An important paper in the area of non-functional requirements is the paper on quality perception by Garvin. Garvin [20] defined five views on software quality from different domains. He states that, because quality is a complex and multifaceted concept, multiple viewpoints are needed to analyze it. The *transcendental view* sees quality as something that can be recognized but impossible to be defined objectively for everybody. The *user view* defines quality as the fit to the user's needs. The *manufacturing view* defines quality as adherence to specification. The *product view* defines quality from the internal characteristics of a product. The *value-based view* defines quality as a trade-off between quality and cost. This thesis is most interested in the characteristics of the product itself. This corresponds to the product view of Garvin.

Non-functional requirement models

Different authors and organizations have refined non-functional requirement into more practical models. The most prominent are the classifications by McCall & Matsumoto, Boehm and the ISO/IEC 9126 standard.

McCall and Matsumoto [36] started by identifying the three main views on non-functional requirement: *product revision*, *product transition* and *product operations*. Every view is associated with a couple of requirements that influence the behaviour of the system. Product revision is linked to *maintainability*, *flexibility* and *testability*. Product transition is linked to *portability*, *reusability* and *interoperability*. Product operations is linked to *correctness*, *reliability*, *efficiency*, *integrity* and *usability*. The requirements are refined into quality criteria. The criteria are used as in indicator of certain requirements. A high traceability, completeness and consistency are indicators that the software has a high degree of correctness. Some criteria are indicators of multiple requirements. Modularity is an important factor for all the requirements that have to do with maintenance and

development because modularity impacts the implementation. To be able to test an application for the criteria McCall and Matsumoto defined metrics for measurement of each specific criteria (see Figure A.1).

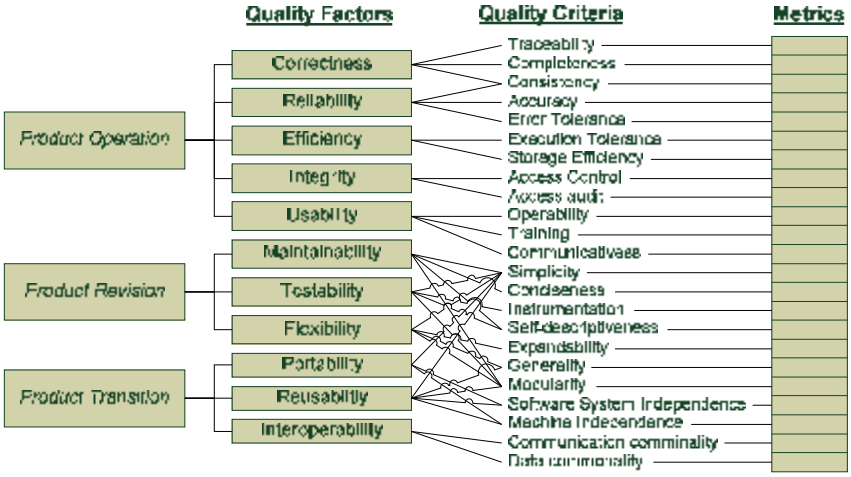


Figure A.1: McCall and Matsumoto's quality model (image from [11])

Boehm [7] divided non-functional requirements on a high level in three basic uses: as-is utility, maintainability and portability. This high level division matches closely with the three views by McCall and Matsumoto. The three high-levels are refined into seven non-functional requirements: *portability, reliability, efficiency, human engineering (usability), testability, understand ability* and *modifiability*. On a lower level, the requirements are refined into characteristics. Certain characteristics are used for multiple requirements. These characteristics are linked to certain metrics that are used to measure and analyze the characteristics.

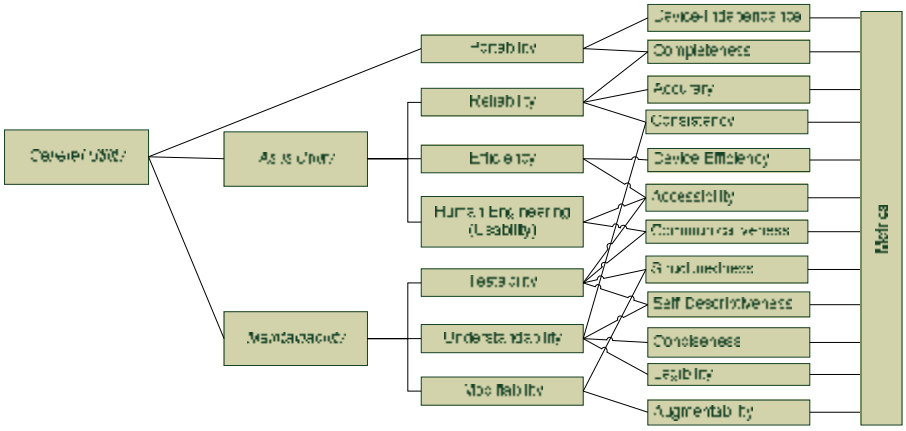


Figure A.2: Boehm's quality model (image from [11])

The International Organization for Standardization (ISO) created an international standard for software quality the ISO/IEC 9126 standard [4]. It was originally published as ISO/IEC 9126:1991 in 1991. It consisted of six quality characteristics and information about how to apply them. It was criticized for being incomplete and incomprehensive and revised in later years. The original model is now split in two parts. The ISO/IEC 9216-X series documents ISE standard. The quality model is documented in ISO/IEC 9126-1:2001. The external metrics of the model are handled ISO/IEC 9126-2:2003, and internal metrics are handled in ISO/IEC 9126-3:2003. The last part, ISO/IEC 9126-4:2004, documents quality in use. The quality evaluation process is now part of ISO/IEC 14598-X series.

Similar to the quality classification by Boehm and McCall, the ISO standard is divided in different levels. The main level consists of *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. These aspects are further refined into sub characteristics and both internal and external metrics. A key difference between the ISO standard and the Boehm and McCall model is that the ISO standard is completely hierarchical because sub-characteristics are linked to only one quality characteristic. The other two models allow sub characteristics to indicate multiple quality characteristic. An overview of this standard is given in Figure A.3.

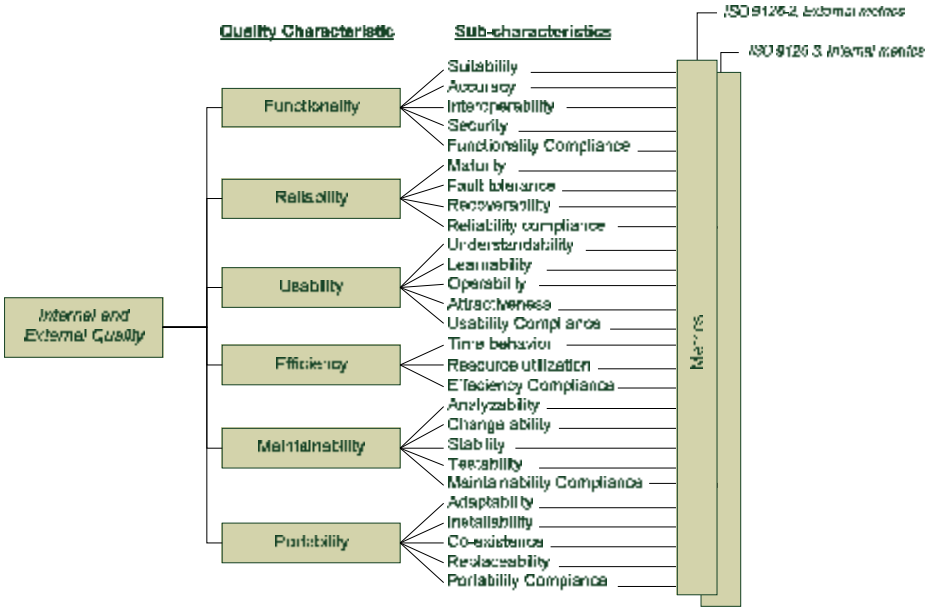


Figure A.3: ISO/IEC 9126-X quality model (image from [11])

When the three main quality classifications are compared, a number of similarities and differences appear. All the models define different levels and link metrics to specific quality criteria. They also share quality characteristics. All models include reliability, efficiency, usability, maintainability and portability. The other quality characteristics are often named differently or put on a different level. McCall has more characteristics on the primary level but they can be traced to the refinement of characteristics of the other models. In the end all three models talk about the same characteristics in more or less specified way. This is also part of the trouble when comparing the classifications because they sometimes name a certain characteristic differently or split the explanation over multiple criteria. For clear and effective communication about a non-functional requirement it is important to have a clear definition of its meaning and applicability.

Non-functional requirement for web applications

Web applications are a special category of IT software. Traditional IT software is focused on applications running on a stand-alone computer. With the development of advanced networks like the World Wide Web a paradigm shift has occurred. Distributed web applications in all kinds of shapes and sizes have appeared. From Amazon.com to shared medical information between hospitals and general practitioners, the networks open up new possibilities for communication. To be able to address the specific needs for development for the web, a new discipline in Software Engineering has emerged, namely *Web Engineering*.

Different authors advocate that development for the web is substantially different from traditional software development and needs tailored processes, models and techniques [3],[13].

[22], [46]. Development for the web also has an impact on the non-functional requirements. The technology, architecture and user interaction require focus on different requirements than those during traditional software development. The defining features of the World Wide Web are that it is available twenty-four hours a day and seven hours a week and accessible by a great number of people from different geographical locations on demand. To make it accessible for all users the interaction has to be made generic to suit all these different users. The Internet is used for both formal and informal communication. Users expect the communication to be secure and reliable. An application that uses the Internet as medium is expected to share the same characteristics. There is a distinction to be made between public and private applications. Restricted private applications are more restricted in terms of users and usability but the important quality requirements remain the same.

Traditional non-functional requirements models only partially describe the context in which web applications should function. Moreover, it is not clear to what extent these models are applicable to modern web applications. Authors have therefore tried to define more web application specific models. Offut [44], [45] identified the seven most important non-functional requirement for web applications, namely *reliability*, *usability*, *security*, *availability*, *scalability*, *maintainability* and *time-to-market*. He identifies the first three as most important. These requirements correspond to the important characteristics of the World Wide Web. One requirement that is not on the list of Offut but is important is *performance*. People expect the application to be safe and available but they also do not want to be kept waiting while performing their task. In a later article he does add performance but groups it with time-to-market. Time-to-market is not a software requirement but a project requirement so this one will not be taken into account. Rudolf and Pirker [18] describe scalability/performance, security, usability and reliability as important non-functional requirements for e-Business. These requirements share the requirement from Offut.

The next part will give a definition of the seven remaining requirements. Different authors and classifications give their own different definitions of these non-functional requirements. This thesis uses a distilled definition from the perspective of web applications:

- **Reliability** is the ability of a system to work correctly when expected. A reliable system does what the users expect it to do, limited within the functionality of the system. To serve the customers on demand web applications require high reliability.
- **Usability** is the ease of use from a user perspective. How long does it take for a user to be able to understand and use basic and advanced features? The users come from a wide range of backgrounds usability is an issue for web applications.
- **Security** is the safety of the network, computer and data. In web applications private data passes through a number of computers, mediums and interfaces and must be protected on different levels. To ensure the safety of data security considerations are needed.
- **Availability** is the accessibility of an application for use. The Internet is available 24/7 and users expect web applications to also be usable 24/7. Web applications also should be accessible from different browsers, platforms and technology from which users access the application. The variety of computers and time-independent access make availability critical for web applications.
- **Scalability** is the ease with which a system can be expanded to serve more users and/or work. Any web application can potentially be linked to a very large number of users. To be able to cope with a growing number of users and/or work a scalable web application is needed in today's environment.

- **Maintainability** is the ease with which parts of the software can be added, removed and changed. Web applications typically have a frequent update rate. The need for high reliability and availability make good support for maintenance critical for success.
- **Performance** is most often used to describe the time it takes to fulfil a specific task or a set of related tasks. It is also used to describe other resources used like memory and storage space. Especially in a web environment where customers can choose between suppliers a low performance will decrease competitive advantage.

Reliability and availability are closely related. If an application is reliable, it means it will not fail and thus is available. Usability is an important part of the software but is often not part of the component framework. The presentation layer is often independent from the other layers and is custom made for every iteration of the components. This leaves reliability, availability, security, scalability, maintainability and performance as important non-functional requirements for web applications.

Not all non-functional requirements are evenly important during design and implementation. Some applications will need a greater focus on security while other focus on usability. Online banking systems have a high demand for security because they want to minimize the risk of fraud and misuse. Digital stores like Amazon.com want to be accessible even for first-time users so they will focus on usability. During product development there are multiple trade-off decisions needed between the non-functional requirements. This means during development a trade-off decisions between different non-functional requirements has to be made. And the more requirements have to be taken into account the greater the complexity of the application. The spider-diagram shows two different implementations of an application, red and yellow. For each axis a decision has to be made about how important that specific non-functional requirements is. The red one focuses on performance, scalability and maintainability and availability and less on the other three. The yellow one has a different focus on usability, availability and reliability and a medium focus on the others. The size of the surface can be seen as a reflection of its complexity. The yellow one wants to focus on almost all the non-functional requirements and this is something that is hard to accomplish (eg. complex). The red has a smaller surface which means less focus on all the different requirements and will be easier implement.

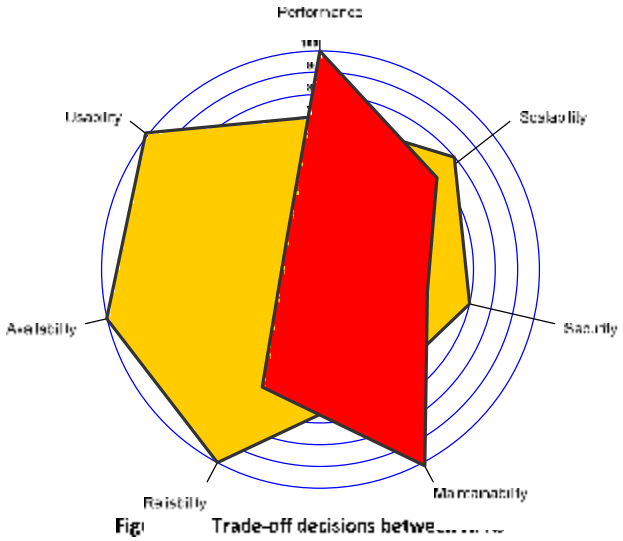


Figure Trade-off decisions betwe...

One of these ways is to model these different choices is using the Non-fuctional requirement framework.

The NFR framework

A framework for evaluation of different non-functional requirements is the NFR framework [12]. The NFR framework is used to express NFRs explicitly which helps in dealing with them in a more rational manner. The goal is to help developers think about the impact different NFRs have on the design and help in evaluating different implementation options and trade-offs. The NFR framework is composed of several, not necessarily sequential, steps:

1. Acquiring or accessing knowledge about
    a. The particular domain and the system which is being developed
    b. Functional requirements for the particular system
    c. Particular kinds of NFRs and associated development techniques
2. Identifying particular NFRs for the domain
3. Decomposing NFRs
4. Identifying "operationalizations" (possible design alternatives for meeting NFRs in the target system)
5. Dealing with:
    a. Ambiguities
    b. Tradeoffs and priorities
    c. Interdependencies among NFRs and operationalizations
6. Selecting operationalizations
7. Supporting decisions with design rationale
8. Evaluating the impact of decisions

The results of these steps are visualized in softgoal interdependency graphs (SIG). The visual representation helps in quickly communicating different design choices and alternatives. The framework is useful in helping to clear up ambiguities and promote the notion and importance of NFRs in software development. The problem with the NFR framework is that it works best for small projects. For a big project the number of choices and alternatives that would have to be mapped is simply too big. A graph with too many lines becomes unreadable, which in turns removes the goal of creating unambiguities. Another problem is that the choices and alternatives are limited by the imagination of the creator. There are no best practices or standardized alternatives and if the users of the framework do not model certain choices they are not considered in the evaluation.

Appendix B: Topicus Case Study Information

This appendix shows an example of different code metrics that can be extracted from source code. In the following section more detailed information from a number of test is given:

1. Server logs
2. Expert interviews
3. Load tests
4. Profiler
5. Software metrics
6. Dependency Structure Matrix/Graph

Appendix B1: Logfile information

The following table is the information gained from the server logs of application A

URL	Count (#)	Avg. time (sec)	Total (min)	Min. time (sec)	Max time (sec)	Standard deviation (sec)
/WF/A/Filtering.aspx	14880	2,52	616	0,00	1223,83	17,12
/WF/A/DocTypen.aspx	41695	0,85	560	0,01	892,73	5,62
/WF/A/PersonalData.aspx	47639	0,68	557	0,02	641,30	5,58
/WF/A/CustomerSearch.aspx	46502	0,71	551	0,03	625,57	3,03
/WF/TP/AanvraagSnelToets.aspx	25852	0,96	412	0,00	1175,57	9,31
/WF/TP/ResultaatAlternatief.aspx	28027	0,86	400	0,01	1184,57	8,47
/WF/A/Workspace.aspx	60640	0,38	388	0,02	611,51	3,11
/WF/TP/AanvraagOnderpandGeerDepot.aspx	13150	1,65	373	0,01	607,69	6,89
/WF/A/Offer.aspx	32018	0,69	370	0,02	289,63	2,47
/WF/A/UploadScanDocument.aspx	2702	6,32	285	0,27	187,67	8,53
/WF/A/BKRtoets.aspx	14582	1,21	284	0,02	84,78	2,46
/WF/TP/Resultaat.aspx	18806	0,84	256	0,00	653,26	7,62
/WF/TP/Default.aspx	79643	0,16	213	0,00	292,50	1,68
/WF/A/AanvraagOnderpandGeenDepot.aspx	2627	2,88	126	0,02	620,49	14,64
/WF/A/Incomes.aspx	9409	0,67	107	0,02	488,37	5,31
/WF/Login.aspx	43655	0,14	106	0,00	1120,60	8,97
/WF/TP/AanvraagKlantGegevens.aspx	33865	0,18	106	0,00	89,75	1,11
/WF/A/RealEstate.aspx	11195	0,56	105	0,01	45,52	1,85
/WF/TP/ResultaatGrijs.aspx	2808	1,45	87	0,01	473,50	11,32
/WF/A/CustomerAction.aspx	7189	0,65	86	0,01	35,26	1,80
/WF/TP/AanvraagOnderpandDepot.aspx	3212	1,56	86	0,01	44,57	3,88
/WF/A/AanvraagLening.aspx	6154	0,83	85	0,20	32,30	0,79
/WF/TP/Afwijzing.aspx	7475	0,63	82	0,01	77,19	1,87
/WF/A/DocumentViewer.aspx	1333	3,00	67	0,02	46,87	5,07
/WF/A/AanvraagKlantGegevens.aspx	11757	0,32	63	0,01	42,62	1,00

Table A.1: Complete top 25 from the server logs of application A

Appendix B2: Log test information

The load test was done with 1, 25, 50, 75 and 100 concurrent users. The scenarios was recorded with Visual Studio 2008. Each load test was done two times. One lasting 10 minutes and one lasting 15 minutes. Fifty percent of the users had a simulated T3 connection and other fifty percent used a T1 connection. This was done not the push the LAN connection speed to its limit and make a more realistic simulation. The load test was done on a local acceptation server of that specific application.

Max User Load	1	25	50	75	100
Request/Sec	1,255	7,675	8,91	10,17	10,615
Request Failed (x10)	0	0	0	2	2,7
Requests Cached Percentage (x10)	6,285	5,35	4,965	4,57	4,085
Avg. Response Time (sec)	0,29	2,83	5,145	6,66	8,215
Avg. Content Length (bytes) (x1000)	12,7495	10,1875	9,4845	9,0065	9,5065
Tests/Sec (x0,1)	0,2	9,15	7,15	8,3	0
Tests Failed	0	0	0	10	0
Avg. Test Time (sec) (x100)	4,24	2,495	4,75	7,125	0
Avg. Transaction Time (sec)	0	0	0	0	0
Avg. Page Time (sec)	0,375	4,86	9,75	15	0

Table A.2: Load test result

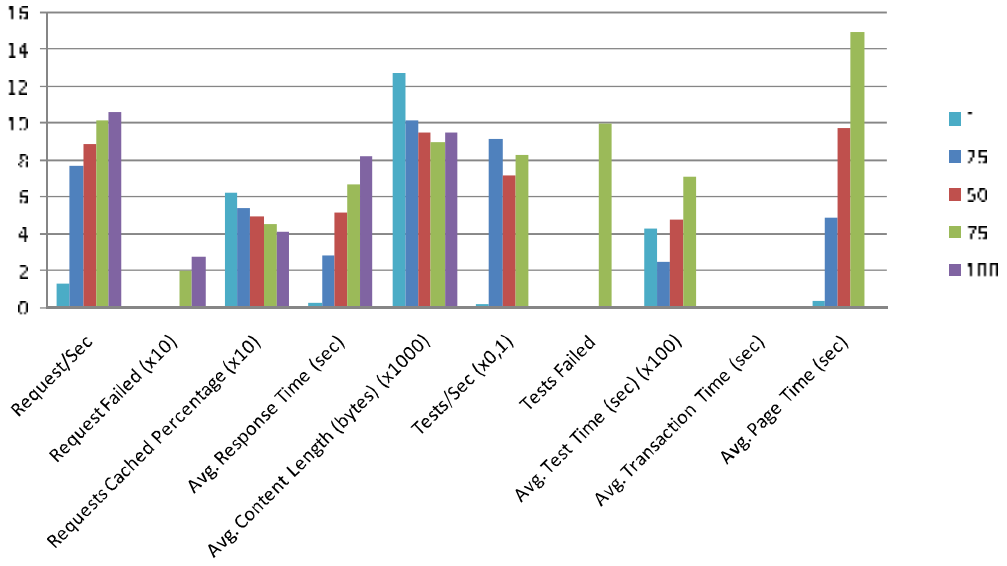


Figure A.5: Load test summary

URL	Average page time with 1 user (sec)	Page count with 1 user (x10)	Average page time with 25 users (sec)	Page count with 25 users (x10)	Average page time with 50 users (sec)	Page count with 50 users (x10)	Average page time with 75 users (sec)	Page count with 75 users (x10)
/WF/A/Fiattering.aspx	2,415	3,6	24,6	21,3	43,85	15,6	74,7	22,5
/WF/A/CustomerSearch.aspx	2,005	3	15,75	14,2	36,85	5,45	53,35	15
/WF/A/Workspace.aspx	0,88	3	8,05	14,2	17,4	5,45	29,25	15
/WebResource.axd	0,125	31,5	3,455	149,1	8,635	109,2	12,6	157,5
/WF/A/CustomerSearchDialog.aspx	0,185	1,5	3,34	7,1	5,875	5,2	3,85	7,5
/WF/A/Default.aspx	0,027	1,5	2,385	7,1	3,625	5,2	5,88	7,5
/WF/Login.aspx	0,635	4,5	2,9	21,3	4,475	15,6	6,14	22,1
/WF/A/PersonalData.aspx	0,605	1,5	2,335	7,1	2,52	5,2	3,99	7,5
/	0,144	1,5	12,405	7,1	0,725	29,95	1,165	7
/TL/EmptyIframe.html	0,017	16,5	0,055	32,56	0,0725	57,2	0,043	82,5

Table A.3: Load Test result per page

Appendix B3: Profiler

The profiler runs of the fiattering of both application A and B is shown in the following tables. For simplicity only the top of the call tree is shown.

Method	Percentage (%)	Sub-Methods	Percentage (%)
Fiattering.btn_voorlopigeFiattering_Click	77,68%	GenerateAllDocuments	75,94%
		Fiatteren	17,30%
		RefreshRisicoKlasse	4,84%
		GetRisicoKlasseForKlantPropositie	0,50%
		GetBKRScoreForKlantPropositie	0,31%

Table A.4: Highest part of the call tree taken for the profiler run of application A

Method	Percentage (%)	Sub-Methods	Percentage (%)
Fiattering.btn_voorlopigeFiattering_Click	31,20%	GenerateAllDocuments	51,62%
		Fiatteer	47,99%
		RefreshFiatteringRules	0,39%

Table A.5: Highest part of the call tree taken for the profiler run of application B

Appendix B1: Software Metrics

The coloured modules are the ones that are used by the flattering module. There was no comment information available for the FORCE and TopicusLibrary so these are not counted. The coloured cells are the ones that are used by the flattering the application.

Application A	# Types	# Abstract Types	# IL instruction	# lines of code	# lines of comment	% comment	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
FMS	671	36	260.931	35.576	43.452	0,54	0	370	4,53	1	0,05	0,05
FMS.Berekeningen	16	3	2.765	397	555	0,58	4	4	1,88	0,5	0,19	0,31
FMS.HDN	189	11	19.603	3.368	4.080	0,54	6	97	4,08	0,94	0,06	0
FMS.Jobs	1	0	47	9	20	0,68	0	6	1	1	0	0
FMS.Koppelingen	10	1	8.215	1.244	570	0,31	14	43	0,8	0,75	0,1	0,15
FMS.Koppelingen.NHG	146	14	15.501	2.276	1.679	0,42	3	35	1,62	0,92	0,1	0,02
FMS.Koppelingen.TaxatieAanvraag	7	0	2.950	435	342	0,44	1	18	1,57	0,95	0	0,05
FMS.Logic	7	1	9.705	1.322	755	0,36	135	93	1,29	0,41	0,14	0,45
FMS.Objects	453	52	102.831	15.781	9.522	0,37	470	42	4,27	0,08	0,11	0,8
FMS.Producten	42	14	9.401	1.353	495	0,26	92	40	2,81	0,3	0,33	0,36
FMS.Templates	4	0	1.176	171	128	0,42	0	8	1	1	0	0
Average per module	*141*	*12*	*39.375*	*5.630*	*5.600*	*0,45*	*66*	*69*	*2,26*	*0,71*	*0,10*	*0,20*
Average per module used by assembly	*200*	*18*	*65.641*	*9.279*	*9.225*	*0,40*	*119*	*99*	*2,60*	*0,51*	*0,15*	*0,35*
Difference	*59*	*6*	*26.266*	*3.649*	*3.625*	*-4,39%*	*53*	*30*	*0,34*	*-0,21*	*0,06*	*0,15*

Table A.6: Software metrics overview of application A

Application B	# Types	# Abstract Types	# IL instruction	# lines of code	# lines of comment	% comment	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
FMS	291	10	132.050	18.629	0	0%	0	325	5,71	1	0,03	0,03
FMS.Berekeningen	19	4	3.225	470	0	0%	7	5	1,89	0,42	0,21	0,37
FMS.Documents	185	4	56.963	7.783	0	0%	6	83	2,51	0,93	0,02	0,05
FMS.HDN	75	10	22.966	3.277	0	0%	1	195	5,77	0,99	0,13	0,13
FMS.HDN.XSD	350	0	9.147	2.719	0	0%	53	0	1,76	0	0	1
FMS.Koppelingen	8	1	6.596	995	0	0%	1	22	1	0,96	0,12	0,08
FMS.Koppelingen.NHG	171	17	18.008	2.715	0	0%	4	54	1,79	0,93	0,1	0,03
FMS.Logic	33	2	23.608	3.271	0	0%	112	187	1,61	0,63	0,06	0,31
FMS.Objects	441	56	77.616	11.706	0	0%	422	40	4,3	0,09	0,13	0,79
FMS.POP3Service	11	2	1.190	181	0	0%	0	5	1,27	1	0,18	0,18
FMS.Producten	55	22	12.249	1.810	0	0%	100	69	3,22	0,41	0,4	0,19

Average per module	*149*	*12*	*33.056*	*4.869*	*0*	*0,0%*	*64*	*90*	*2,80*	*0,67*	*0,13*	*0,29*
Average per module used by flattering	*141*	*16*	*42.557*	*6.147*	*0*	*0*	*107*	*108*	*3*	*1*	*0*	*0*
Difference	*-8*	*4*	*9.501*	*1.278*	*0*	*0,00%*	*43*	*18*	*0,15*	*-0,08*	*0,03*	*0,01*

Table A.7: Software metrics overview of application B

The following tables show information about the source code of application A and B. Multiple tools were used that calculate the different metrics proposed in this thesis. These source code metrics are especially usable to see if the code is reusable.

Total	LOC	Comments	Comment (%)	Lines	Files	Cyclomatic Complexity
Application A	199.628	37.580	15,84%	264.790	1.437	19.790
Application B	148.070	28.360	16,07%	192.929	1.280	13.679
Difference application A -> B	51.558	9.220	0,23%	71.861	157	6.111
Difference application A -> B (%)	34,82%	32,51%		37,25%	12,27%	
Equality application A -> B (%)	65,18%	67,49%		62,75%	87,73%	

Table A.8: Software metrics comparison

Application A -> Application B	LOC	Comment	Lines	Files
New lines of code in newer project	73.260	16.063	105.822	352
Removed lines of code from older project	124.818	-25.283	-177.683	509
Equal lines in both projects	57.002	10.113	67.115	502
Inline changes from A to B	17.808	2.184	19.992	426
Total for new version	148.070	28.360	192.929	1.280
Direct reuse in newer project (%) (only equal)	38,50%	35,66%	34,79%	39,22%
Percentage changed (%)	61,50%	64,34%	65,21%	60,78%

Table A.9: Detailed code metric comparison

Appendix B5: Dependency structure Matrix / Graph

The top-right half of the matrix should be read as follows:
The number is the number of methods from the horizontal module using methods from the vertical module
The bottom-left half of the matrix should be read the other way around:
The number is the number of methods from the horizontal being used by methods from the vertical module

Modules of application A		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
FMS	**1**		3	31	2	69	200	33	4			1375		15	15	5		766	1295
FMS.HDN	**2**	3					19					62						1	19
FMS.Koppelingen.NHG	**3**	137					5					96							5
FMS.Koppelingen.TaxatieAanvraag	**4**	9					1					8							1
FMS.Logic	**5**	38					16	11				84	5	1	2	12			15
FMS.Producten	**6**	144	25	7	3	21						90	2					0	74
FMS.Koppelingen	**7**	25				6			2			53				22			59
FORCE.Koppelingen.LogiLink.VIS	**8**	10						9											13
FMS.Jobs	**9**											2							2
FMS.Templates	**10**											3							10
FMS.Objects	**11**	1937	232	187	44	275	81	124		2	2		1	58	14	4		3	763
FMS.Berekeningen	**12**					6	3					1							4
FORCE.Constants	**13**	14				2						3							8
FORCE.Flatteren	**14**	22				6						39							17
FORCE.Koppelingen.Logilink.BKR	**15**	55				23		53				9							3
FORCE.Koppelingen.LogiLink	**16**																		
TopicusLibrary.Webcontrols	**17**	146	1									0							45
TopicusLibrary	**18**	133	15	4	1	9	31	21	9	3	9	56	4	8	8	13		50	

Table A.10: Dependency structure matrix (DSM) of application A

Application A -> B	#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FMS	1		-2	-23	30	38	33	0	-423	0	11	-5	-1	0	-156	-219
FMS.HDN	2	1		-3	0	-26	0	0	40	0	0	0	0	0	-1	-8
FMS.Logic	3	28	8		-4	-70	11	-4	189	8	1	15	22	0	0	0
FMS.Koppelingen.NHG	4	-136	0	-4		-8	0	0	12	0	0	0	1	0	0	4
FMS.Producten	5	0	67	-57	-1		0	0	43	2	0	0	0	0	0	46
FMS.Koppelingen	6	-25	0	6	0	0		2	-15	0	0	0	-22	0	0	-6
FORCE.Koppelingen.LogiLink.MS	7	-3	0	-18	0	0	9		0	0	0	0	0	0	0	1
FMS.Objects	8	-656	181	-195	-89	-36	25	0		-1	25	-2	5	0	0	-281
FMS.Berekeningen	9	0	0	-12	0	-1	0	0	-1		0	0	0	0	0	0
FORCE.Constants	10	6	0	-1	0	0	0	0	4	0		0	0	0	0	12
FORCE.Flatteren	11	-4	0	-45	0	0	0	0	-32	0	0		0	0	0	1
FORCE.Koppelingen.LogiLink.BKR	12	-5	0	-50	-3	0	53	0	8	0	0	0		0	0	0
FORCE.Koppelingen.Logilink	13	0	0	0	0	0	0	0	0	0	0	0	0		0	0
TopicusLibrary.Webcontrols	14	-38	-1	0	0	0	0	0	-1	0	0	0	0	0		2
TopicusLibrary	15	-31	-8	-12	-6	1	3	0	3	0	1	-1	0	0	4	

Table A.11: Dependency comparison between applications

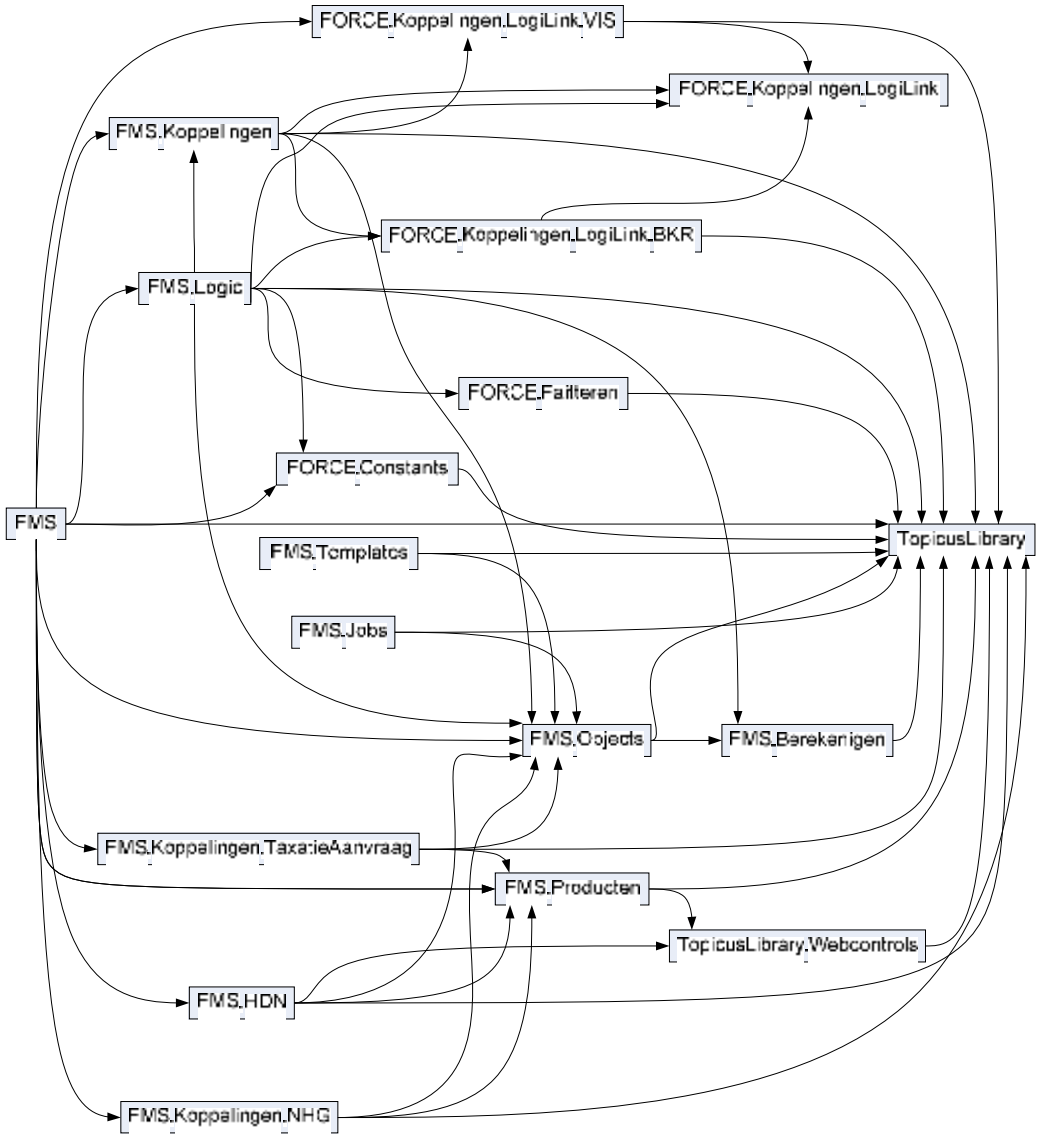


Figure A.6: Dep... e ...ph of application A